Apache Spark 的设计与实现

Yourtion



目錄

Introduction

本文主要讨论 Apache Spark 的设计与实现，重点关注其设计思想、运行原理、实现架构及性能调优，附带讨论与 Hadoop MapReduce 在设计与实现上的区别。不喜欢将该文档称之为“源码分析”，因为本文的主要目的不是去解读实现代码，而是尽量有逻辑地，从设计与实现原理的角度，来理解 job 从产生到执行完成的整个过程，进而去理解整个系统。

讨论系统的设计与实现有很多方法，本文选择 问题驱动 的方式，一开始引入问题，然后分问题逐步深入。从一个典型的 job 例子入手，逐渐讨论 job 生成及执行过程中所需要的系统功能支持，然后有选择地深入讨论一些功能模块的设计原理与实现方式。也许这样的方式比一开始就分模块讨论更有主线。

本文档面向的是希望对 Spark 设计与实现机制，以及大数据分布式处理框架深入了解的 Geeks。

因为 Spark 社区很活跃，更新速度很快，本文档也会尽量保持同步，文档号的命名与 Spark 版本一致，只是多了一位，最后一位表示文档的版本号。

由于技术水平、实验条件、经验等限制，当前只讨论 Spark core standalone 版本中的核心功能，而不是全部功能。诚邀各位小伙伴们加入进来，丰富和完善文档。

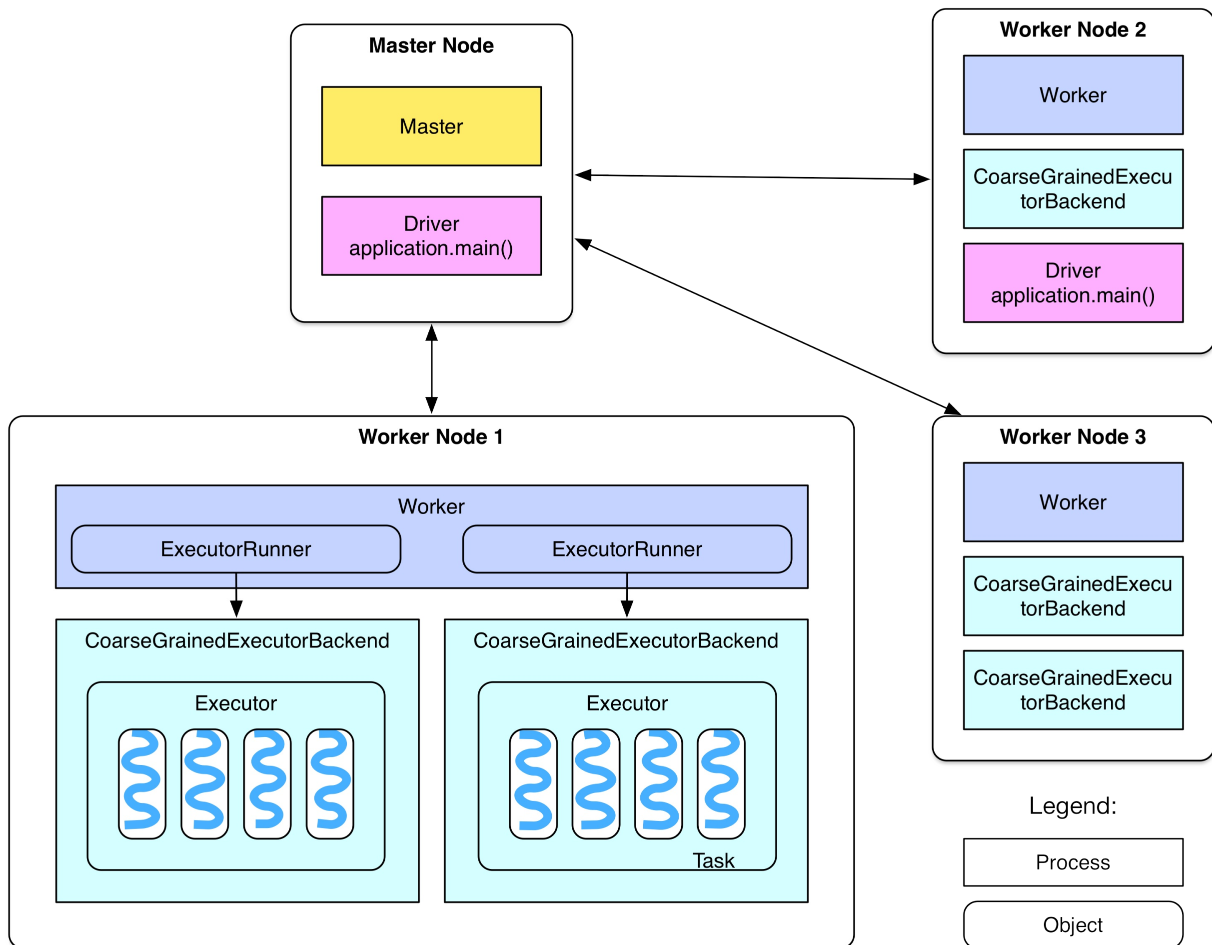
关于学术方面的一些讨论可以参阅相关的论文以及 Matei 的博士论文，也可以看看我之前写的这篇 blog。

好久没有写这么完整的文档了，上次写还是三年前在学 Ng 的 ML 课程的时候，当年好有激情啊。这次的撰写花了 20+ days，从暑假写到现在，大部分时间花在 debug、画图和琢磨怎么写上，希望文档能对大家和自己都有所帮助。

概览

拿到系统后，部署系统是第一件事，那么系统部署成功以后，各个节点都启动了哪些服务？

部署图



从部署图中可以看到

- 整个集群分为 Master 节点和 Worker 节点，相当于 Hadoop 的 Master 和 Slave 节点。
- Master 节点上常驻 Master 守护进程，负责管理全部的 Worker 节点。
- Worker 节点上常驻 Worker 守护进程，负责与 Master 节点通信并管理 executors。
- Driver 官方解释是 “The process running the main() function of the application and creating the SparkContext”。Application 就是用户自己写的 Spark 程序（driver program），比如 WordCount.scala。如果 driver program 在 Master 上运行，比如在 Master 上运行

  ```
  ./bin/run-example SparkPi 10
  ```

  那么 SparkPi 就是 Master 上的 Driver。如果是 YARN 集群，那么 Driver 可能被调度到 Worker 节点上运行（比如上图中的 Worker Node 2）。另外，如果直接在自己的 PC 上运行 driver program，比如在 Eclipse 中运行 driver program，使用

```
val sc = new SparkContext("spark://master:7077", "AppName")
```

去连接 master 的话，driver 就在自己的 PC 上，但是不推荐这样的方式，因为 PC 和 Workers 可能不在一个局域网，driver 和 executor 之间的通信会很慢。

- 每个 Worker 上存在一个或者多个 ExecutorBackend 进程。每个进程包含一个 Executor 对象，该对象持有一个线程池，每个线程可以执行一个 task。
- 每个 application 包含一个 driver 和多个 executors，每个 executor 里面运行的 tasks 都属于同一个 application。
- 在 Standalone 版本中，ExecutorBackend 被实例化成 CoarseGrainedExecutorBackend 进程。

> 在我部署的集群中每个 Worker 只运行了一个 CoarseGrainedExecutorBackend 进程，没有发现如何配置多个 CoarseGrainedExecutorBackend 进程。（应该是运行多个 applications 的时候会产生多个进程，这个我还没有实验，）
>
> 想了解 Worker 和 Executor 的关系详情，可以参阅 @OopsOutOfMemory 同学写的 Spark Executor Driver资源调度小结 。

- Worker 通过持有 ExecutorRunner 对象来控制 CoarseGrainedExecutorBackend 的启停。

了解了部署图之后，我们先给出一个 job 的例子，然后概览一下 job 如何生成与运行。

Job 例子

我们使用 Spark 自带的 examples 包中的 GroupByTest，假设在 Master 节点运行，命令是

```
/* Usage: GroupByTest [numMappers] [numKVPairs] [valSize] [numReducers] */

bin/run-example GroupByTest 100 10000 1000 36
```

GroupByTest 具体代码如下

```
package org.apache.spark.examples

import java.util.Random

import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._

/**
  * Usage: GroupByTest [numMappers] [numKVPairs] [valSize] [numReducers]
  */
object GroupByTest {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("GroupBy Test")
    var numMappers = 100
    var numKVPairs = 10000
    var valSize = 1000
    var numReducers = 36

    val sc = new SparkContext(sparkConf)

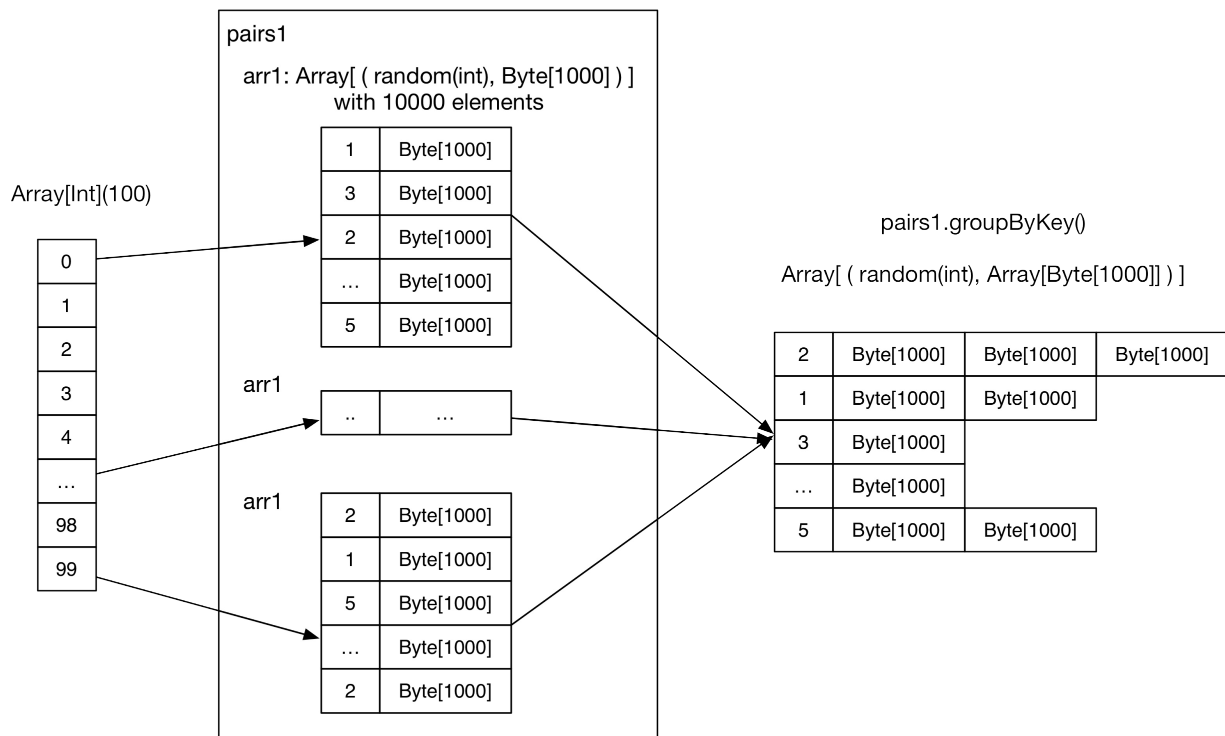
    val pairs1 = sc.parallelize(0 until numMappers, numMappers).flatMap { p =>
      val ranGen = new Random
      var arr1 = new Array[(Int, Array[Byte])](numKVPairs)
      for (i <- 0 until numKVPairs) {
        val byteArr = new Array[Byte](valSize)
        ranGen.nextBytes(byteArr)
        arr1(i) = (ranGen.nextInt(Int.MaxValue), byteArr)
      }
      arr1
```

```
    }.cache
    // Enforce that everything has been calculated and in cache
    pairs1.count

    println(pairs1.groupByKey(numReducers).count)

    sc.stop()
  }
}
```

阅读代码后，用户头脑中 job 的执行流程是这样的：



具体流程很简单，这里来估算下 data size 和执行结果：

1. 初始化 SparkConf()。
2. 初始化 numMappers=100, numKVPairs=10,000, valSize=1000, numReducers= 36。
3. 初始化 SparkContext。这一步很重要，是要确立 driver 的地位，里面包含创建 driver 所需的各种 actors 和 objects。
4. 每个 mapper 生成一个 `arr1: Array[(Int, Byte[])]`，length 为 numKVPairs。每一个 Byte[] 的 length 为 valSize，Int 为随机生成的整数。`Size(arr1) = numKVPairs * (4 + valSize) = 10MB`，所以 `Size(pairs1) = numMappers * Size(arr1) = 1000MB`。这里的数值计算结果都是约等于。
5. 每个 mapper 将产生的 arr1 数组 cache 到内存。
6. 然后执行一个 action 操作 count()，来统计所有 mapper 中 arr1 中的元素个数，执行结果是 `numMappers * numKVPairs = 1,000,000`。这一步主要是为了将每个 mapper 产生的 arr1 数组 cache 到内存。
7. 在已经被 cache 的 paris1 上执行 groupByKey 操作，groupByKey 产生的 reducer （也就是 partition） 个数为 numReducers。理论上，如果 hash(Key) 比较平均的话，每个 reducer 收到的 record 个数为 `numMappers * numKVPairs / numReducer = 27,777`，大小为 `Size(pairs1) / numReducer = 27MB`。
8. reducer 将收到的 `<Int, Byte[]>` records 中拥有相同 Int 的 records 聚在一起，得到 `<Int, list(Byte[], Byte[], ..., Byte[])>`。
9. 最后 count 将所有 reducer 中 records 个数进行加和，最后结果实际就是 pairs1 中不同的 Int 总个数。

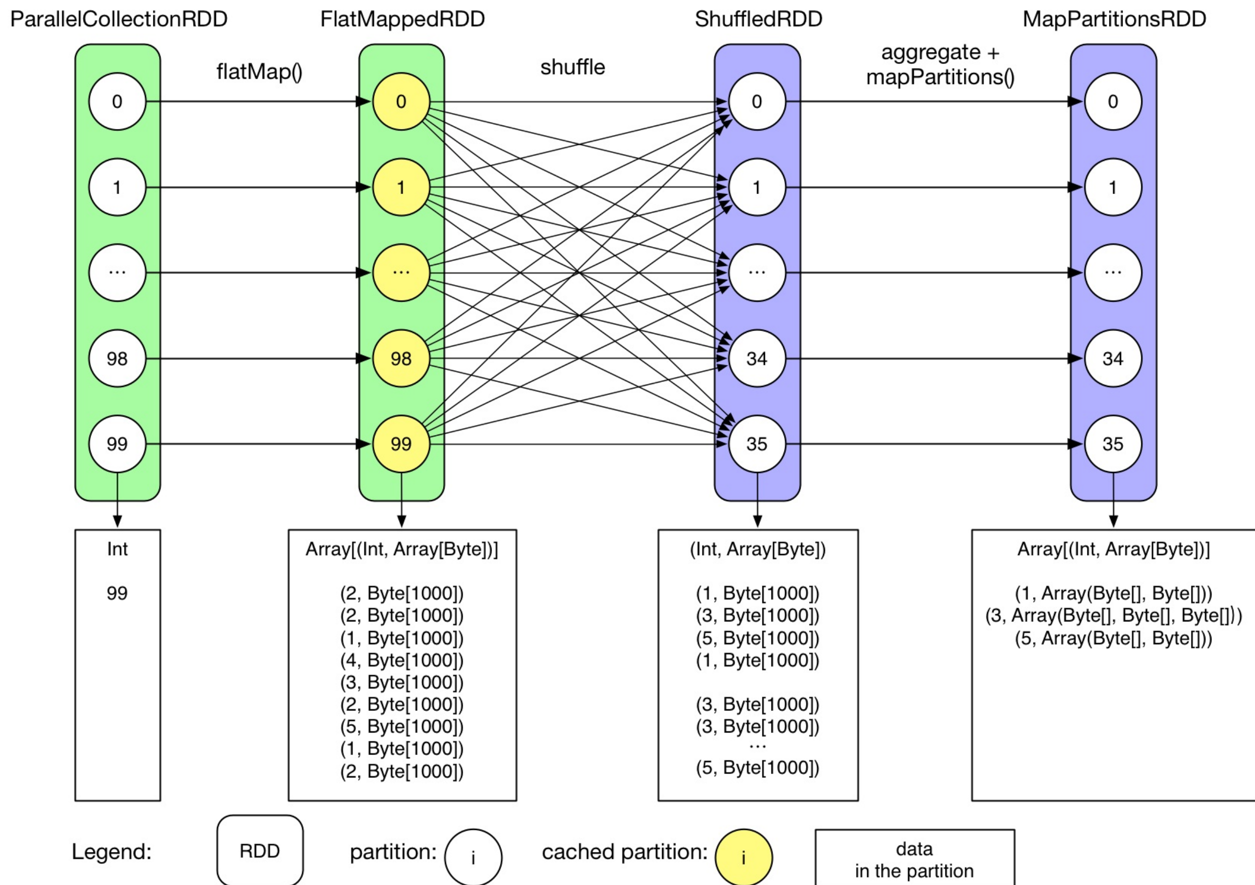
Job 逻辑执行图

Job 的实际执行流程比用户头脑中的要复杂，需要先建立逻辑执行图（或者叫数据依赖图），然后划分逻辑执行图生成 DAG 型的物理执行图，然后生成具体 task 执行。分析一下这个 job 的逻辑执行图：

使用 `RDD.toDebugString` 可以看到整个 logical plan （RDD 的数据依赖关系）如下

```
MapPartitionsRDD[3] at groupByKey at GroupByTest.scala:51 (36 partitions)
  ShuffledRDD[2] at groupByKey at GroupByTest.scala:51 (36 partitions)
    FlatMappedRDD[1] at flatMap at GroupByTest.scala:38 (100 partitions)
      ParallelCollectionRDD[0] at parallelize at GroupByTest.scala:38 (100 partitions)
```

用图表示就是：



> 需要注意的是 data in the partition 展示的是每个 partition 应该得到的计算结果，并不意味着这些结果都同时存在于内存中。

根据上面的分析可知：

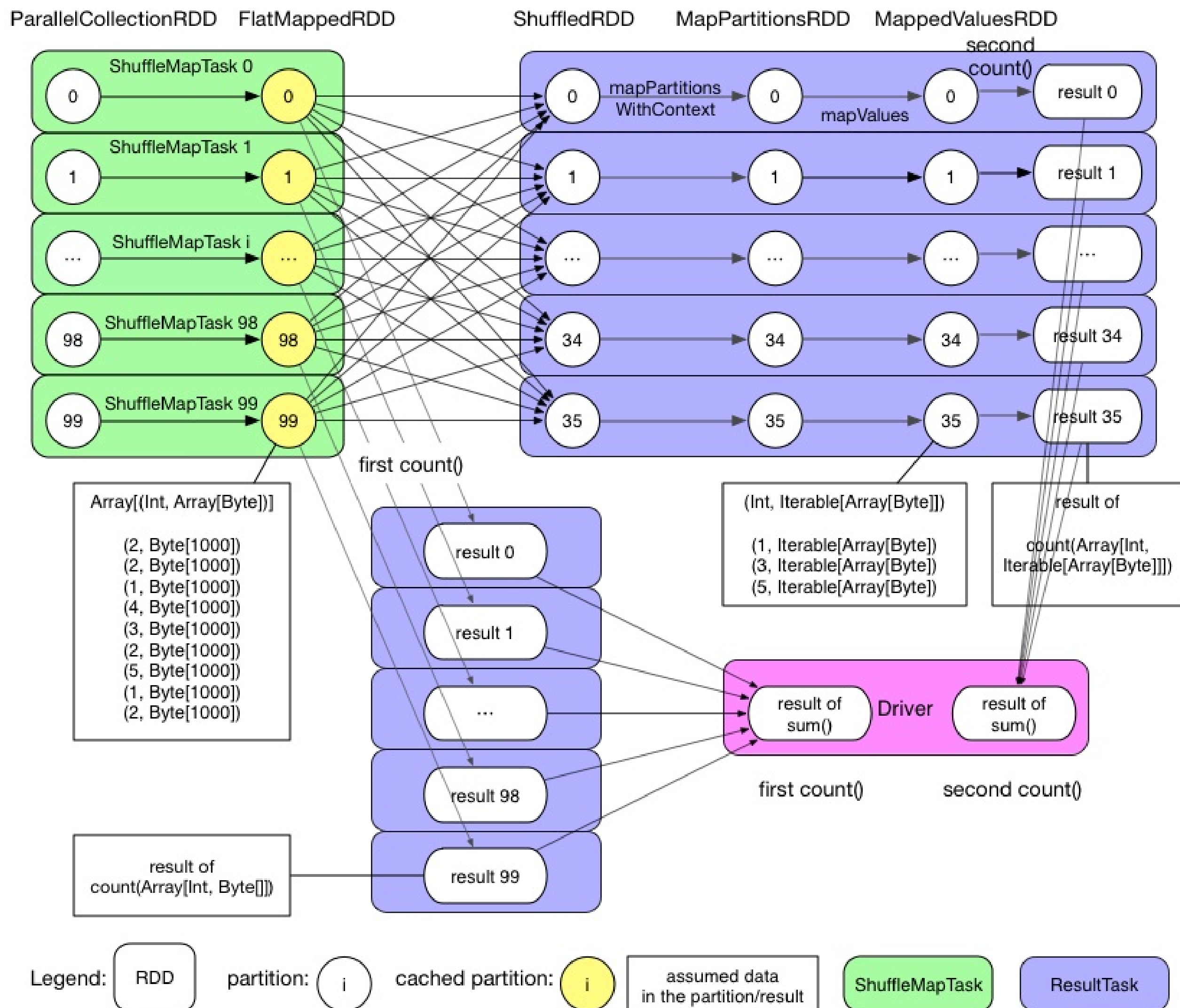
- 用户首先 init 了一个0-99 的数组： `0 until numMappers`
- parallelize() 产生最初的 ParrallelCollectionRDD，每个 partition 包含一个整数 i。
- 执行 RDD 上的 transformation 操作（这里是 flatMap）以后，生成 FlatMappedRDD，其中每个 partition 包含一个 Array[(Int, Array[Byte])]。
- 第一个 count() 执行时，先在每个 partition 上执行 count，然后执行结果被发送到 driver，最后在 driver 端进行 sum。
- 由于 FlatMappedRDD 被 cache 到内存，因此这里将里面的 partition 都换了一种颜色表示。
- groupByKey 产生了后面两个 RDD，为什么产生这两个在后面章节讨论。
- 如果 job 需要 shuffle，一般会产生 ShuffledRDD。该 RDD 与前面的 RDD 的关系类似于 Hadoop 中 mapper 输出数据与 reducer 输入数据之间的关系。
- MapPartitionsRDD 里包含 groupByKey() 的结果。
- 最后将 MapPartitionsRDD 中的 每个value（也就是Array[Byte]）都转换成 Iterable 类型。
- 最后的 count 与上一个 count 的执行方式类似。

可以看到逻辑执行图描述的是 **job 的数据流：job 会经过哪些 transformation()，中间生成哪些 RDD 及 RDD 之间的依赖关系。**

Job 物理执行图

逻辑执行图表示的是数据上的依赖关系，不是 task 的执行图。在 Hadoop 中，用户直接面对 task，mapper 和 reducer 的职责分明：一个进行分块处理，一个进行 aggregate。Hadoop 中 整个数据流是固定的，只需要填充 map() 和 reduce() 函数即可。Spark 面对的是更复杂的数据处理流程，数据依赖更加灵活，很难将数据流和物理 task 简单地统一在一起。因此 Spark 将数据流和具体 task 的执行流程分开，并设计算法将逻辑执行图转换成 task 物理执行图，转换算法后面的章节讨论。

针对这个 job，我们先画出它的物理执行 DAG 图如下：



可以看到 GroupByTest 这个 application 产生了两个 job，第一个 job 由第一个 action（也就是 `pairs1.count`）触发产生，分析一下第一个 job：

- 整个 job 只包含 1 个 stage（不明白什么是stage没关系，后面章节会解释，这里只需知道有这样一个概念）。
- Stage 0 包含 100 个 ResultTask。
- 每个 task 先计算 flatMap，产生 FlatMappedRDD，然后执行 action() 也就是 count()，统计每个 partition 里 records 的个数，比如 partition 99 里面只含有 9 个 records。
- 由于 pairs1 被声明要进行 cache，因此在 task 计算得到 FlatMappedRDD 后会将其包含的 partitions 都 cache 到 executor 的内存。
- task 执行完后，driver 收集每个 task 的执行结果，然后进行 sum()。
- job 0 结束。

第二个 job 由 `pairs1.groupByKey(numReducers).count` 触发产生。分析一下该 job：

- 整个 job 包含 2 个 stage。
- Stage 1 包含 100 个 ShuffleMapTask，每个 task 负责从 cache 中读取 pairs1 的一部分数据并将其进行类似 Hadoop 中 mapper 所做的 partition，最后将 partition 结果写入本地磁盘。
- Stage 0 包含 36 个 ResultTask，每个 task 首先 shuffle 自己要处理的数据，边 fetch 数据边进行 aggregate 以及后续的 mapPartitions() 操作，最后进行 count() 计算得到 result。
- task 执行完后，driver 收集每个 task 的执行结果，然后进行 sum()。
- job 1 结束。

可以看到物理执行图并不简单。与 MapReduce 不同的是，Spark 中一个 application 可能包含多个 job，每个 job 包含多个 stage，每个 stage 包含多个 task。怎么划分 **job**，怎么划分 **stage**，怎么划分 **task** 等等问题会在后面的章节介绍。

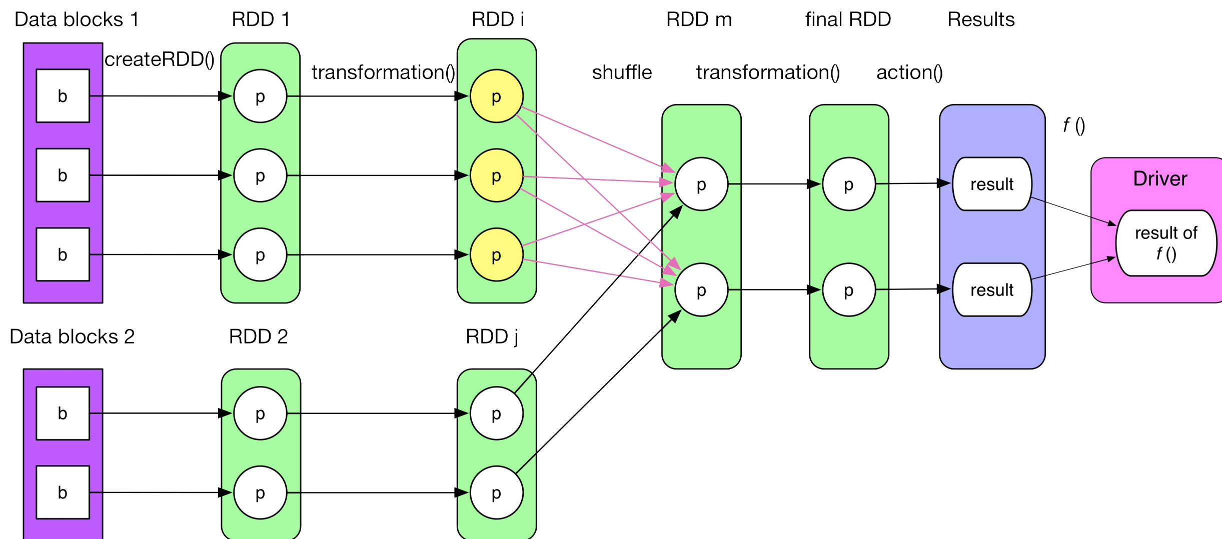
Discussion

到这里，我们对整个系统和 job 的生成与执行有了概念，而且还探讨了 cache 等特性。 接下来的章节会讨论 job 生成与执行涉及到的系统核心功能，包括：

1. 如何生成逻辑执行图
2. 如何生成物理执行图
3. 如何提交与调度 Job
4. Task 如何生成、执行与结果处理
5. 如何进行 shuffle
6. cache机制
7. broadcast 机制

Job 逻辑执行图

General logical plan



典型的 Job 逻辑执行图如上所示，经过下面四个步骤可以得到最终执行结果：

- 从数据源（可以是本地 file，内存数据结构， HDFS，HBase 等）读取数据创建最初的 RDD。上一章例子中的 parallelize() 相当于 createRDD()。
- 对 RDD 进行一系列的 transformation() 操作，每一个 transformation() 会产生一个或多个包含不同类型 T 的 RDD[T]。T 可以是 Scala 里面的基本类型或数据结构，不限于 (K, V)。但如果是 (K, V)，K 不能是 Array 等复杂类型（因为难以在复杂类型上定义 partition 函数）。
- 对最后的 final RDD 进行 action() 操作，每个 partition 计算后产生结果 result。
- 将 result 回送到 driver 端，进行最后的 f(list[result]) 计算。例子中的 count() 实际包含了action() 和 sum() 两步计算。

> RDD 可以被 cache 到内存或者 checkpoint 到磁盘上。RDD 中的 partition 个数不固定，通常由用户设定。RDD 和 RDD 之间 partition 的依赖关系可以不是 1 对 1，如上图既有 1 对 1 关系，也有多对多的关系。

逻辑执行图的生成

了解了 Job 的逻辑执行图后，写程序时候会在脑中形成类似上面的数据依赖图。然而，实际生成的 RDD 个数往往比我们想想的个数多。

要解决逻辑执行图生成问题，实际需要解决：

- 如何产生 RDD，应该产生哪些 RDD？
- 如何建立 RDD 之间的依赖关系？

1. 如何产生 RDD，应该产生哪些 RDD？

解决这个问题的初步想法是让每一个 transformation() 方法返回（new）一个 RDD。事实也基本如此，只是某些 transformation() 比较复杂，会包含多个子 transformation()，因而会生成多个 RDD。这就是*实际 RDD 个数比我们想象的多一些* 的原因。

如何计算每个 **RDD** 中的数据？逻辑执行图实际上是 computing chain，那么 transformation() 的计算逻辑在哪里被

perform？每个 RDD 里有 compute() 方法，负责接收来自上一个 RDD 或者数据源的 input records，perform transformation() 的计算逻辑，然后输出 records。

产生哪些 RDD 与 transformation() 的计算逻辑有关，下面讨论一些典型的 transformation() 及其创建的 RDD。官网上已经解释了每个 transformation 的含义。iterator(split) 的意思是 foreach record in the partition。这里空了很多，是因为那些 transformation() 较为复杂，会产生多个 RDD，具体会在下一节图示出来。

Transformation	Generated RDDs	Compute()
map(func)	MappedRDD	iterator(split).map(f)
filter(func)	FilteredRDD	iterator(split).filter(f)
flatMap(func)	FlatMappedRDD	iterator(split).flatMap(f)
mapPartitions(func)	MapPartitionsRDD	f(iterator(split))
mapPartitionsWithIndex(func)	MapPartitionsRDD	f(split.index, iterator(split))
sample(withReplacement, fraction, seed)	PartitionwiseSampledRDD	PoissonSampler.sample(iterator(split)) BernoulliSampler.sample(iterator(split))
pipe(command, [envVars])	PipedRDD	
union(otherDataset)		
intersection(otherDataset)		
distinct([numTasks]))		
groupByKey([numTasks])		
reduceByKey(func, [numTasks])		
sortByKey([ascending], [numTasks])		
join(otherDataset, [numTasks])		
cogroup(otherDataset, [numTasks])		
cartesian(otherDataset)		
coalesce(numPartitions)		
repartition(numPartitions)		

2. 如何建立 RDD 之间的联系？

RDD 之间的数据依赖问题实际包括三部分：

- RDD 本身的依赖关系。要生成的 RDD（以后用 RDD x 表示）是依赖一个 parent RDD，还是多个 parent RDDs？
- RDD x 中会有多少个 partition ？
- RDD x 与其 parent RDDs 中 partition 之间是什么依赖关系？是依赖 parent RDD 中一个还是多个 partition？

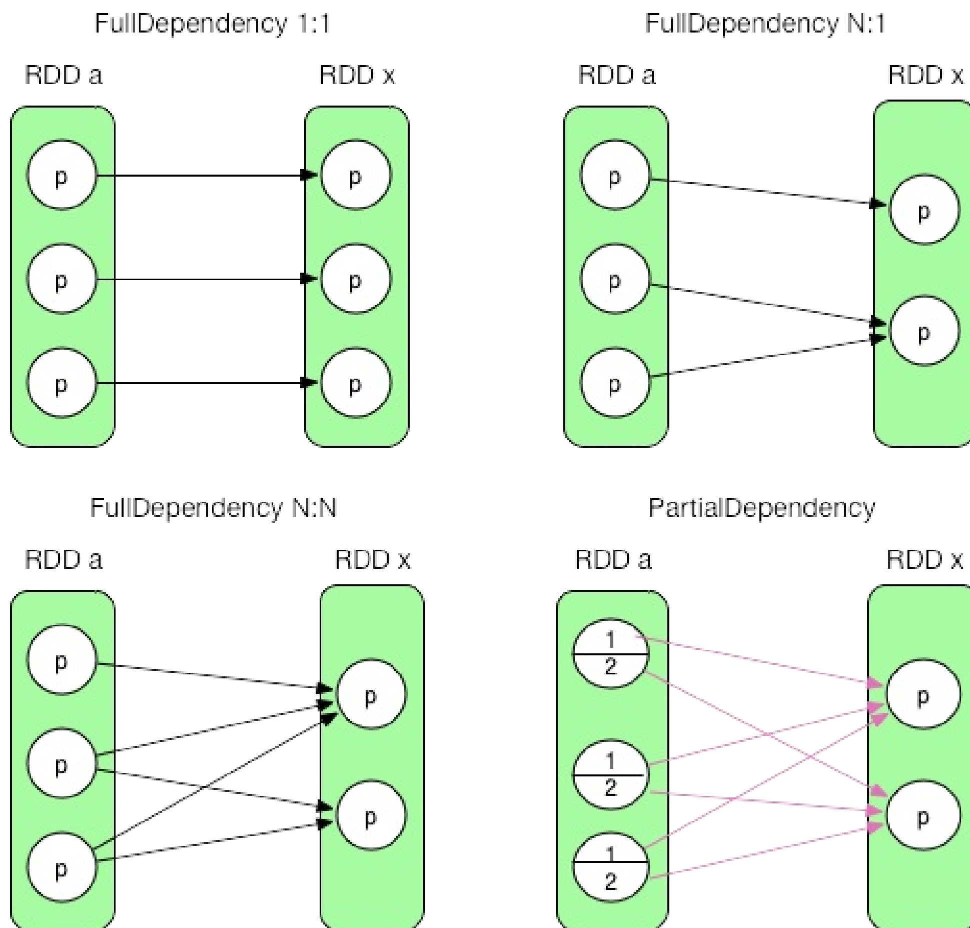
第一个问题可以很自然的解决，比如 `x = rdda.transformation(rddb)` (e.g., x = a.join(b)) 就表示 RDD x 同时依赖于 RDD a 和 RDD b。

第二个问题中的 partition 个数一般由用户指定，不指定的话一般取 `max(numPartitions[parent RDD 1], .., numPartitions[parent RDD n])` 。

第三个问题比较复杂。需要考虑这个 transformation() 的语义，不同的 transformation() 的依赖关系不同。比如 map() 是 1:1，而 groupByKey() 逻辑执行图中的 ShuffledRDD 中的每个 partition 依赖于 parent RDD 中所有的 partition，还有更复杂

的情况。

再次考虑第三个问题，RDD x 中每个 partition 可以依赖于 parent RDD 中一个或者多个 partition。而且这个依赖可以是完全依赖或者部分依赖。部分依赖指的是 parent RDD 中某 partition 中一部分数据与 RDD x 中的一个 partition 相关，另一部分数据与 RDD x 中的另一个 partition 相关。下图展示了完全依赖和部分依赖。



前三个是完全依赖，RDD x 中的 partition 与 parent RDD 中的 partition/partitions 完全相关。最后一个是部分依赖，RDD x 中的 partition 只与 parent RDD 中的 partition 一部分数据相关，另一部分数据与 RDD x 中的其他 partition 相关。

在 Spark 中，完全依赖被称为 NarrowDependency，部分依赖被称为 ShuffleDependency。其实 ShuffleDependency 跟 MapReduce 中 shuffle 的数据依赖相同（mapper 将其 output 进行 partition，然后每个 reducer 会将所有 mapper 输出中属于自己的 partition 通过 HTTP fetch 得到）。

- 第一种 1:1 的情况被称为 OneToOneDependency。
- 第二种 N:1 的情况被称为 N:1 NarrowDependency。
- 第三种 N:N 的情况被称为 N:N NarrowDependency。不属于前两种情况的完全依赖都属于这个类别。
- 第四种被称为 ShuffleDependency。

对于 NarrowDependency，具体 RDD x 中的 partitoin i 依赖 parrent RDD 中一个 partition 还是多个 partitions，是由 RDD x 中的 `getParents(partition i)` 决定（下图中某些例子会详细介绍）。还有一种 RangeDependency 的完全依赖，不过该依赖目前只在 UnionRDD 中使用，下面会介绍。

所以，总结下来 partition 之间的依赖关系如下：

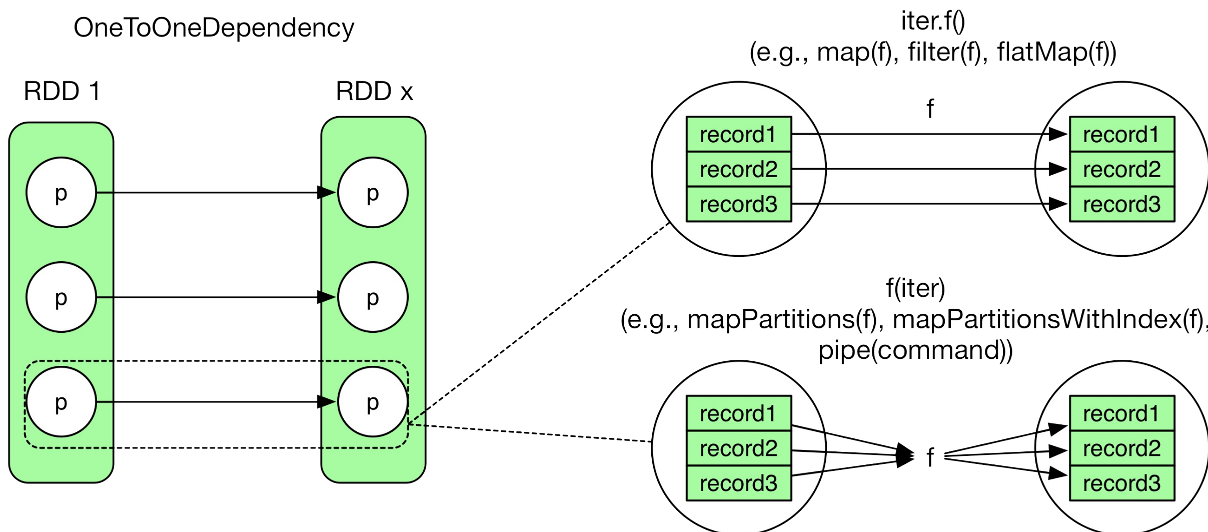
- NarrowDependency (使用黑色实线或黑色虚线箭头表示)
  - OneToOneDependency (1:1)
    - NarrowDependency (N:1)

  - NarrowDependency (N:N)
  - RangeDependency (只在 UnionRDD 中使用)
- ShuffleDependency (使用红色箭头表示)

之所以要划分 NarrowDependency 和 ShuffleDependency 是为了生成物理执行图，下一章会具体介绍。

> 需要注意的是第三种 NarrowDependency (N:N) 很少在两个 RDD 之间出现。因为如果 parent RDD 中的 partition 同时被 child RDD 中多个 partitions 依赖，那么最后生成的依赖图往往与 ShuffleDependency 一样。只是对于 parent RDD 中的 partition 来说一个是完全依赖，一个是部分依赖，而箭头数没有少。所以 Spark 定义的 NarrowDependency 其实是 “each partition of the parent RDD is used by at most one partition of the child RDD“，也就是只有 OneToOneDependency (1:1) 和 NarrowDependency (N:1) 两种情况。但是，自己设计的奇葩 RDD 确实可以呈现出 NarrowDependency (N:N) 的情况。这里描述的比较乱，其实看懂下面的几个典型的 RDD 依赖即可。

如何计算得到 **RDD x** 中的数据（**records**）？下图展示了 OneToOneDependency 的数据依赖，虽然 partition 和 partition 之间是 1:1，但不代表计算 records 的时候也是读一个 record 计算一个 record。 下图右边上下两个 pattern 之间的差别类似于下面两个程序的差别：



code1 of iter.f()

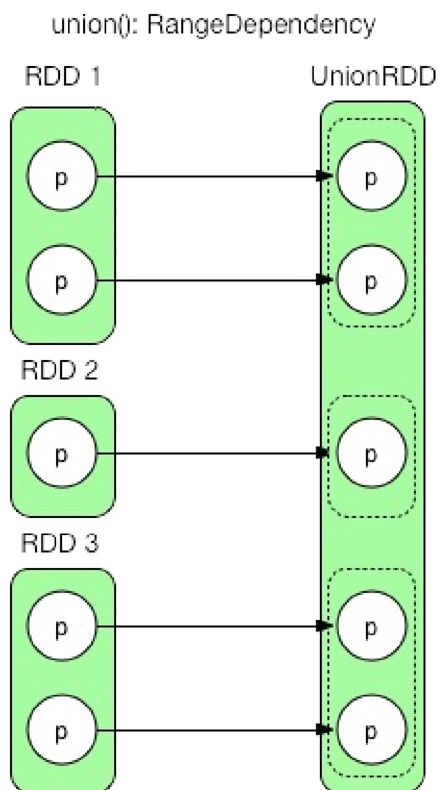
```
int[] array = {1, 2, 3, 4, 5}
for(int i = 0; i < array.length; i++)
    f(array[i])
```

code2 of f(iter)

```
int[] array = {1, 2, 3, 4, 5}
f(array)
```

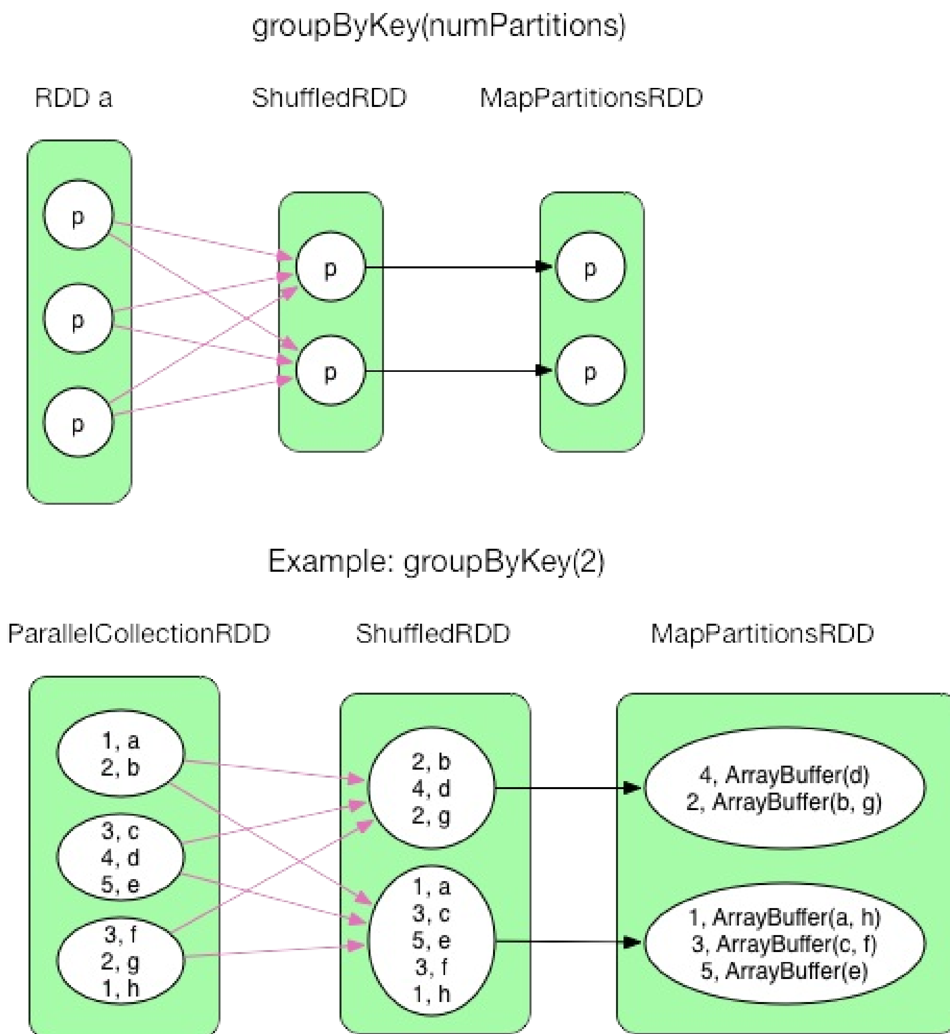
3. 给出一些典型的 transformation() 的计算过程及数据依赖图

1) union(otherRDD)



union() 将两个 RDD 简单合并在一起，不改变 partition 里面的数据。RangeDependency 实际上也是 1:1，只是为了访问 union() 后的 RDD 中的 partition 方便，保留了原始 RDD 的 range 边界。

2) groupByKey(numPartitions)



上一章已经介绍了 groupByKey 的数据依赖，这里算是温故而知新 吧。

groupByKey() 只需要将 Key 相同的 records 聚合在一起，一个简单的 shuffle 过程就可以完成。ShuffledRDD 中的 compute() 只负责将属于每个 partition 的数据 fetch 过来，之后使用 mapPartitions() 操作（前面的 OneToOneDependency 展示过）进行 aggregate，生成 MapPartitionsRDD，到这里 groupByKey() 已经结束。最后为了统一返回值接口，将 value 中的 ArrayBuffer[] 数据结构抽象化成 Iterable[]。

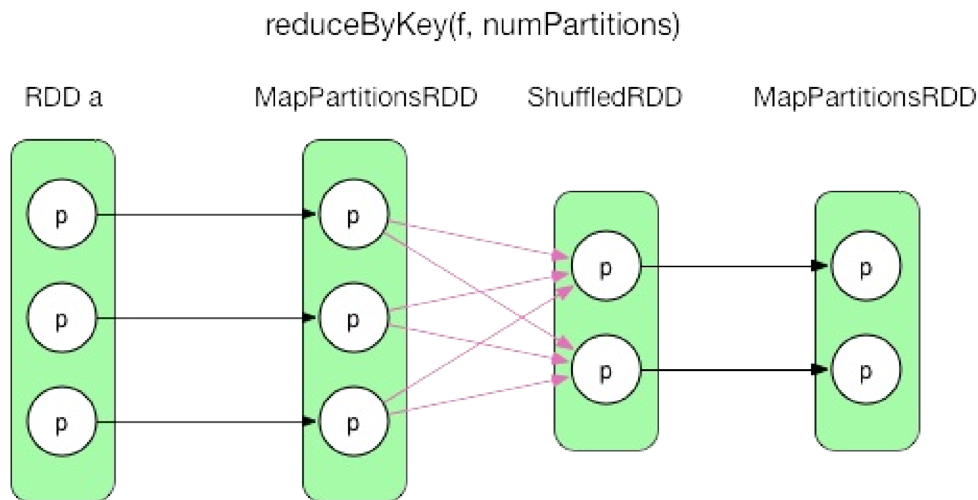
> groupByKey() 没有在 map 端进行 combine，因为 map 端 combine 只会省掉 partition 里面重复 key 占用的空间，当重复 key 特别多时，可以考虑开启 combine。
>
> 这里的 ArrayBuffer 实际上应该是 CompactBuffer，An append-only buffer similar to ArrayBuffer, but more memory-efficient for small buffers.

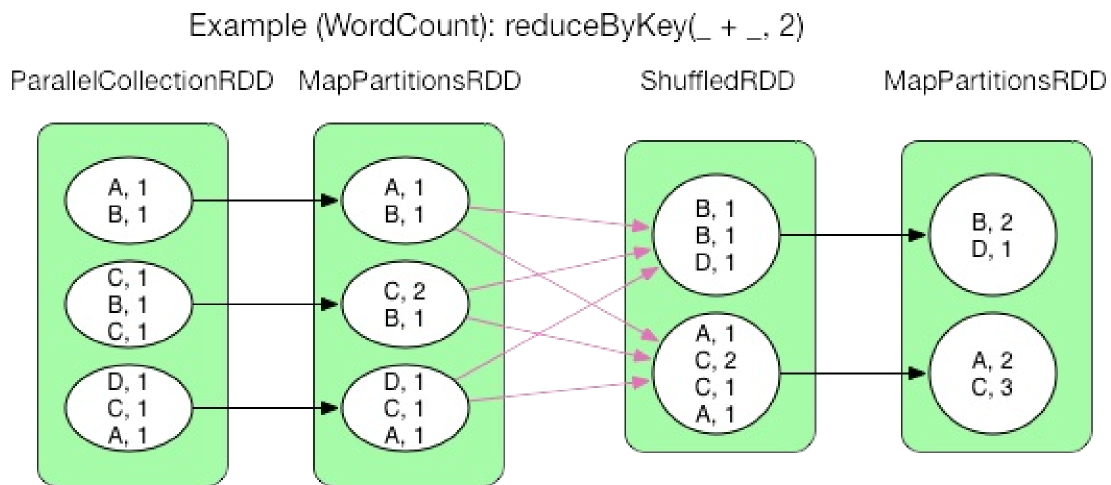
ParallelCollectionRDD 是最基础的 RDD，直接从 local 数据结构 create 出的 RDD 属于这个类型，比如

```
val pairs = sc.parallelize(List(1, 2, 3, 4, 5), 3)
```

生成的 pairs 就是 ParallelCollectionRDD。

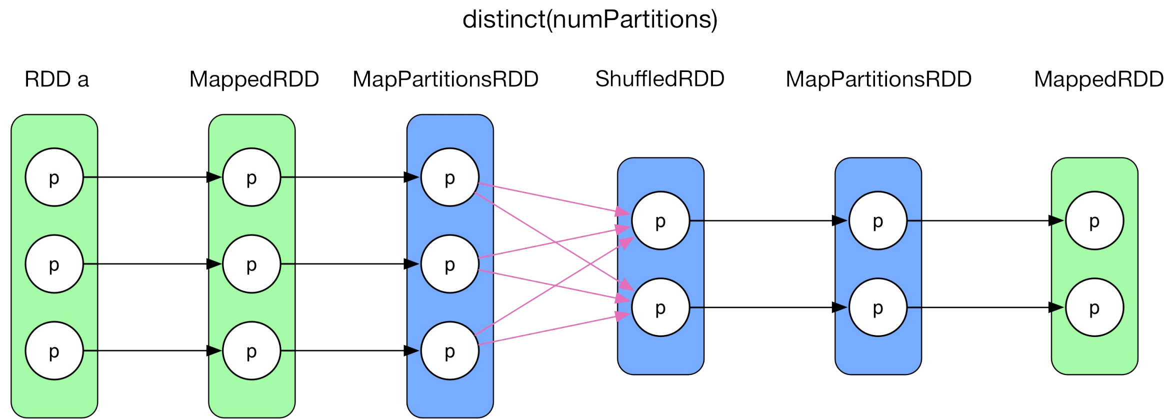
2) reduceyByKey(func, numPartitions)

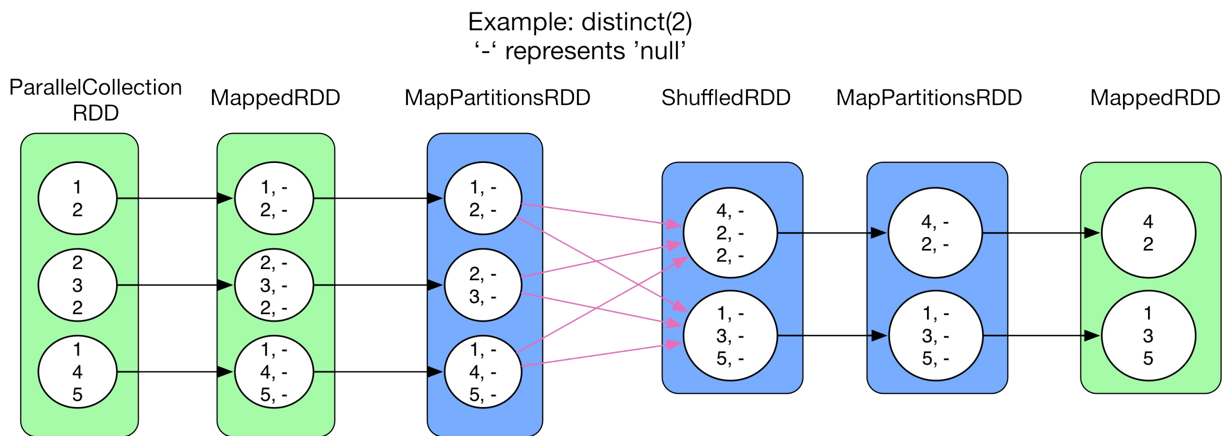




reduceyByKey() 相当于传统的 MapReduce，整个数据流也与 Hadoop 中的数据流基本一样。reduceyByKey() 默认在 map 端开启 combine()，因此在 shuffle 之前先通过 mapPartitions 操作进行 combine，得到 MapPartitionsRDD，然后 shuffle 得到 ShuffledRDD，然后再进行 reduce（通过 aggregate + mapPartitions() 操作来实现）得到 MapPartitionsRDD。

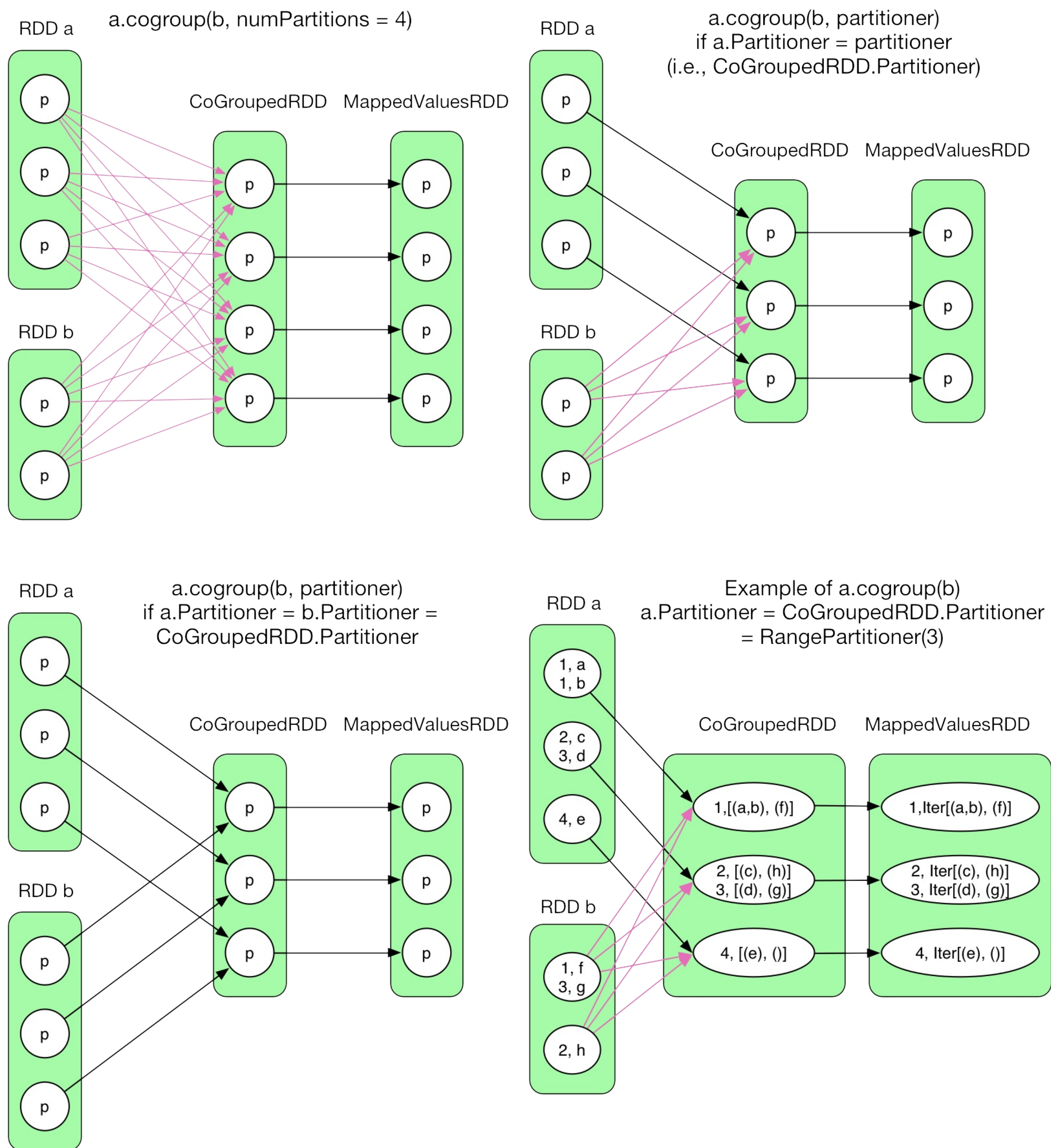
3) distinct(numPartitions)





distinct() 功能是 deduplicate RDD 中的所有的重复数据。由于重复数据可能分散在不同的 partition 里面，因此需要 shuffle 来进行 aggregate 后再去重。然而，shuffle 要求数据类型是 `<K, V>`。如果原始数据只有 Key（比如例子中 record 只有一个整数），那么需要补充成 `<K, null>`。这个补充过程由 map() 操作完成，生成 MappedRDD。然后调用上面的 reduceByKey() 来进行 shuffle，在 map 端进行 combine，然后 reduce 进一步去重，生成 MapPartitionsRDD。最后，将 `<K, null>` 还原成 K，仍然由 map() 完成，生成 MappedRDD。蓝色的部分就是调用的 reduceByKey()。

4) cogroup(otherRDD, numPartitions)



与 groupByKey() 不同，cogroup() 要 aggregate 两个或两个以上的 RDD。那么 **CoGroupedRDD** 与 **RDD a** 和 **RDD b** 的关系都必须是 **ShuffleDependency** 么？是否存在 **OneToOneDependency**？

首先要明确的是 CoGroupedRDD 存在几个 partition 可以由用户直接设定，与 RDD a 和 RDD b 无关。然而，如果 CoGroupedRDD 中 partition 个数与 RDD a/b 中的 partition 个数不一样，那么不可能存在 1:1 的关系。

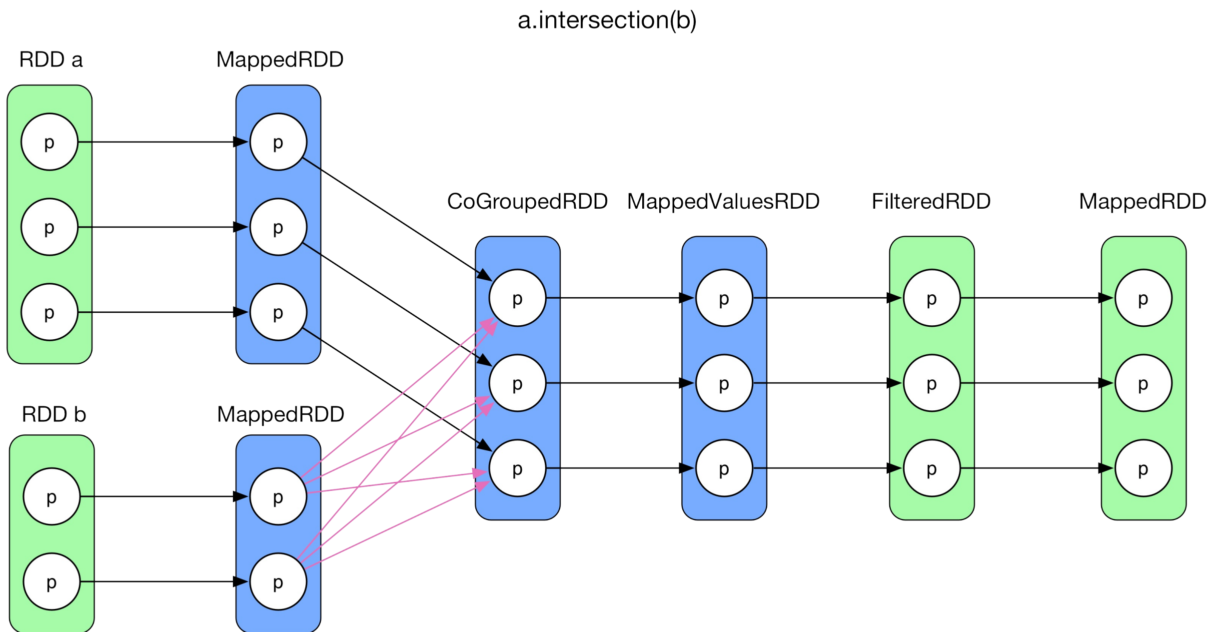
再次，cogroup() 的计算结果放在 CoGroupedRDD 中哪个 partition 是由用户设置的 partitioner 确定的（默认是 HashPartitioner）。那么可以推出：即使 RDD a/b 中的 partition 个数与 CoGroupedRDD 中的一样，如果 RDD a/b 中的 partitioner 与 CoGroupedRDD 中的不一样，也不可能存在 1:1 的关系。比如，在上图的 example 里面，RDD a 是 RangePartitioner，b 是 HashPartitioner，CoGroupedRDD 也是 RangePartitioner 且 partition 个数与 a 的相同。那么很自然地，a 中的每个 partition 中 records 可以直接送到 CoGroupedRDD 中对应的 partition。RDD b 中的 records 必须再次进行划分与 shuffle 后才能进入对应的 partition。

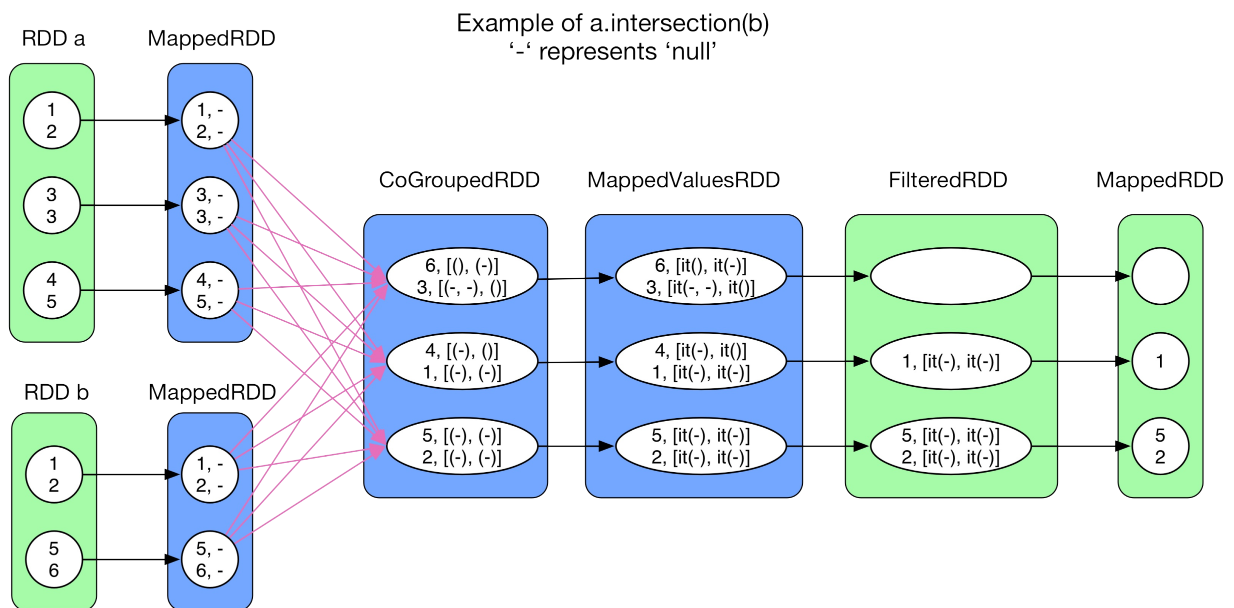
最后，经过上面分析，对于两个或两个以上的 **RDD** 聚合，当且仅当聚合后的 **RDD** 中 **partitioner** 类别及 **partition** 个数与前面的 **RDD** 都相同，才会与前面的 **RDD** 构成 **1:1** 的关系。否则，只能是 **ShuffleDependency**。这个算法对应的代码可以

在 `CoGroupedRDD.getDependencies()` 中找到，虽然比较难理解。

> Spark 代码中如何表示 CoGroupedRDD 中的 partition 依赖于多个 parent RDDs 中的 partitions？
>
> 首先，将 CoGroupedRDD 依赖的所有 RDD 放进数组 rdds[RDD] 中。再次，foreach i，如果 CoGroupedRDD 和 rdds(i) 对应的 RDD 是 OneToOneDependency 关系，那么 Dependecy[i] = new OneToOneDependency(rdd)，否则 = new ShuffleDependency(rdd)。最后，返回与每个 parent RDD 的依赖关系数组 deps[Dependency]。
>
> Dependency 类中的 getParents(partition id) 负责给出某个 partition 按照该 dependency 所依赖的 parent RDD 中的 partitions: List[Int]。
>
> getPartitions() 负责给出 RDD 中有多少个 partition，以及每个 partition 如何序列化。

5) intersection(otherRDD)

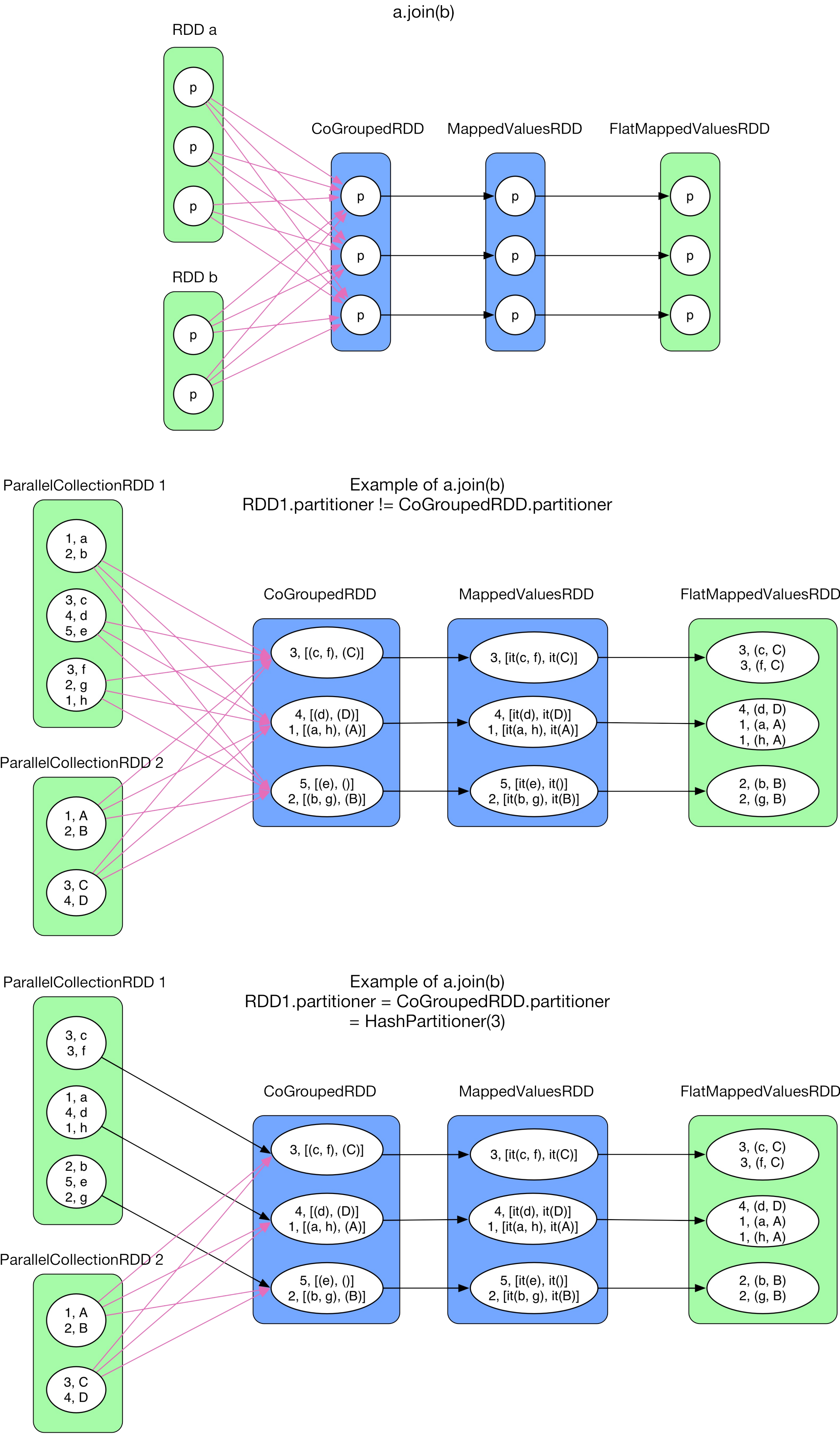




intersection() 功能是抽取出 RDD a 和 RDD b 中的公共数据。先使用 map() 将 RDD[T] 转变成 RDD[(T, null)]，这里的 T 只

要不是 Array 等集合类型即可。接着，进行 a.cogroup(b)，蓝色部分与前面的 cogroup() 一样。之后再使用 filter() 过滤掉 [iter(groupA()), iter(groupB())] 中 groupA 或 groupB 为空的 records，得到 FilteredRDD。最后，使用 keys() 只保留 key 即可，得到 MappedRDD。

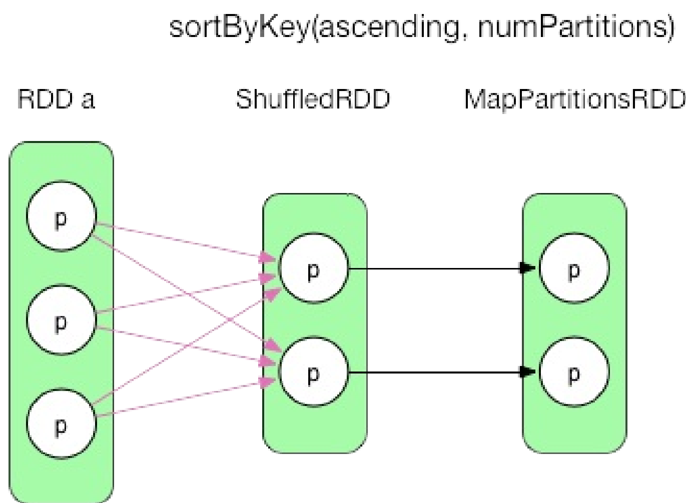
6) **join(otherRDD, numPartitions)**

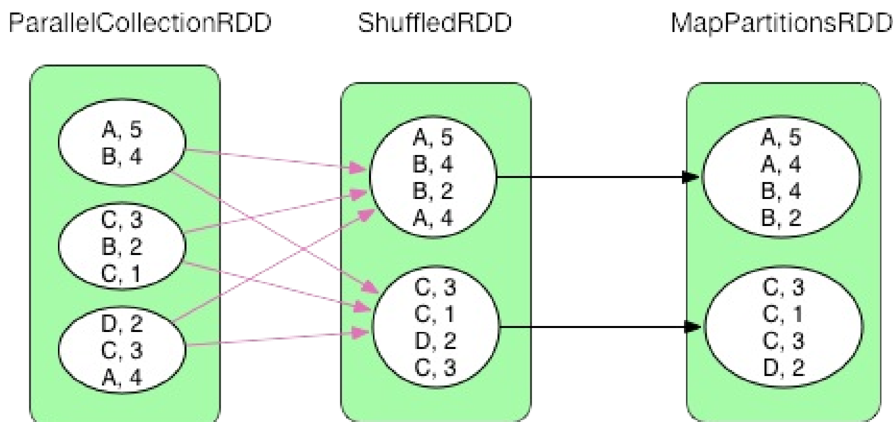

a.join(b)
RDD a
RDD b
CoGroupedRDD
MappedValuesRDD
FlatMappedValuesRDD
p
Example of a.join(b)
RDD1.partitioner != CoGroupedRDD.partitioner
ParallelCollectionRDD 1
ParallelCollectionRDD 2
1, a
2, b
3, c
4, d
5, e
3, f
2, g
1, h
1, A
2, B
3, C
4, D
3, [(c, f), (C)]
4, [(d), (D)]
1, [(a, h), (A)]
5, [(e), ()]
2, [(b, g), (B)]
3, [it(c, f), it(C)]
4, [it(d), it(D)]
1, [it(a, h), it(A)]
5, [it(e), it()]
2, [it(b, g), it(B)]
3, (c, C)
3, (f, C)
4, (d, D)
1, (a, A)
1, (h, A)
2, (b, B)
2, (g, B)
Example of a.join(b)
RDD1.partitioner = CoGroupedRDD.partitioner
= HashPartitioner(3)
3, c
3, f
1, a
4, d
1, h
2, b
5, e
2, g

join() 将两个 RDD[(K, V)] 按照 SQL 中的 join 方式聚合在一起。与 intersection() 类似，首先进行 cogroup()，得到 `<K, (Iterable[V1], Iterable[V2])>` 类型的 MappedValuesRDD，然后对 Iterable[V1] 和 Iterable[V2] 做笛卡尔集，并将集合 flat() 化。

这里给出了两个 example，第一个 example 的 RDD 1 和 RDD 2 使用 RangePartitioner 划分，而 CoGroupedRDD 使用 HashPartitioner，与 RDD 1/2 都不一样，因此是 ShuffleDependency。第二个 example 中， RDD 1 事先使用 HashPartitioner 对其 key 进行划分，得到三个 partition，与 CoGroupedRDD 使用的 HashPartitioner(3) 一致，因此数据依赖是 1:1。如果 RDD 2 事先也使用 HashPartitioner 对其 key 进行划分，得到三个 partition，那么 join() 就不存在 ShuffleDependency 了，这个 join() 也就变成了 hashjoin()。

7) sortByKey(ascending, numPartitions)

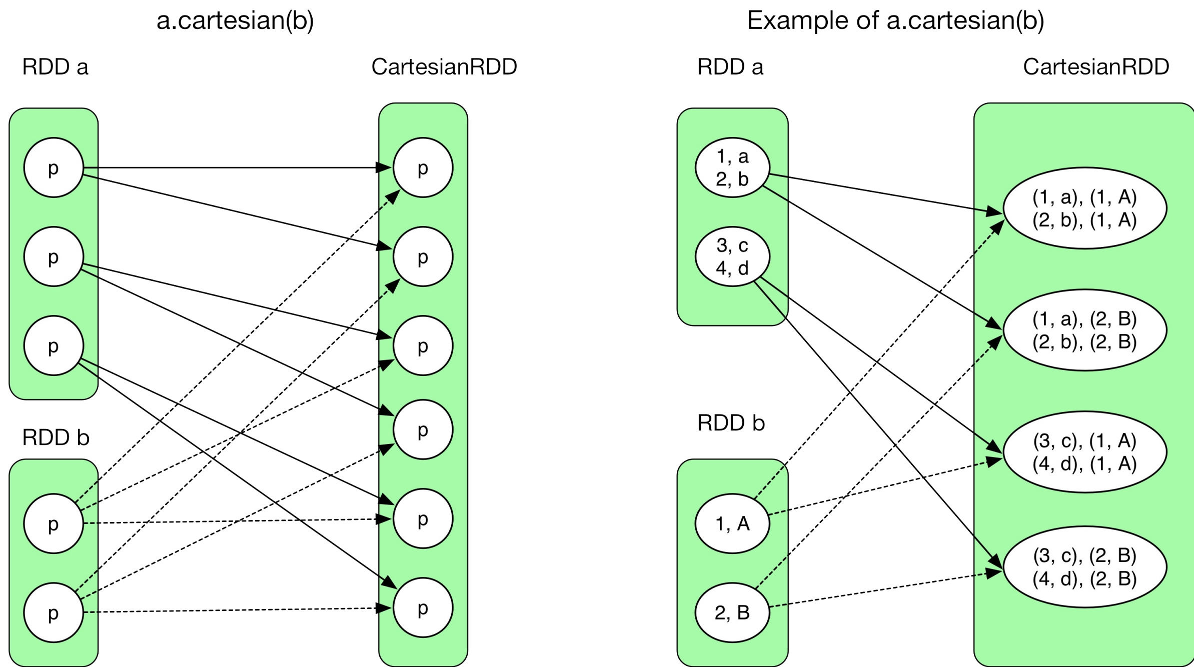




sortByKey() 将 RDD[(K, V)] 中的 records 按 key 排序，ascending = true 表示升序，false 表示降序。目前 sortByKey() 的数据依赖很简单，先使用 shuffle 将 records 聚集在一起（放到对应的 partition 里面），然后将 partition 内的所有 records 按 key 排序，最后得到的 MapPartitionsRDD 中的 records 就有序了。

> 目前 sortByKey() 先使用 Array 来保存 partition 中所有的 records，再排序。

8) cartesian(otherRDD)

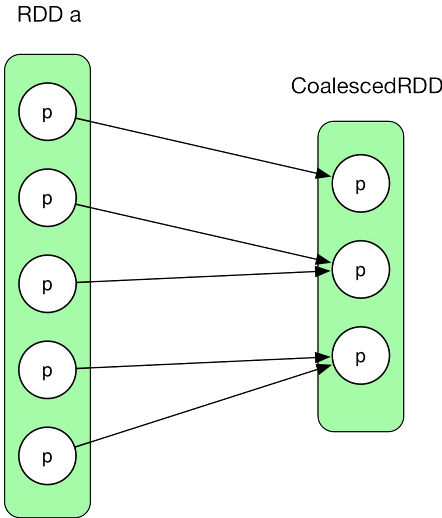


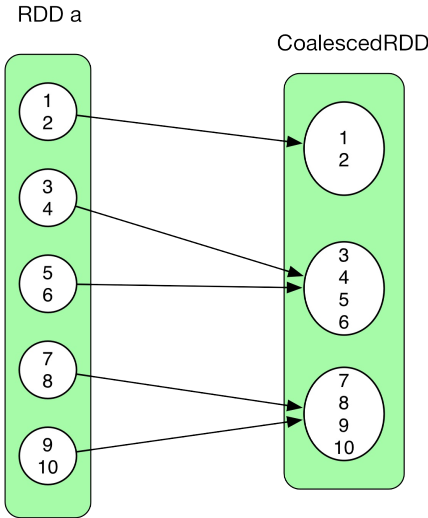
Cartesian 对两个 RDD 做笛卡尔集，生成的 CartesianRDD 中 partition 个数 = partitionNum(RDD a) * partitionNum(RDD b)。

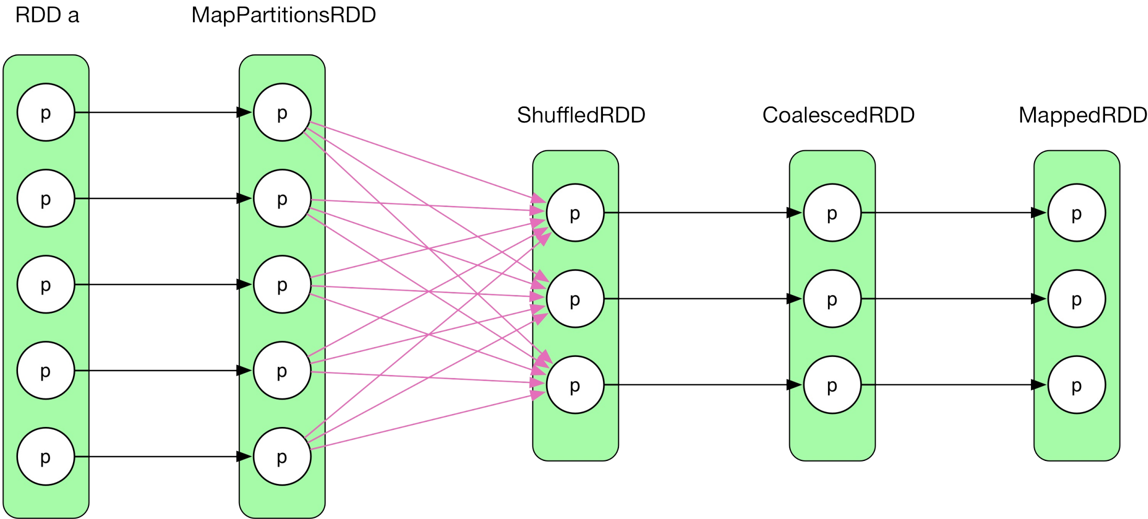
这里的依赖关系与前面的不太一样，CartesianRDD 中每个partition 依赖两个 parent RDD，而且其中每个 partition 完全依赖 RDD a 中一个 partition，同时又完全依赖 RDD b 中另一个 partition。这里没有红色箭头，因为所有依赖都是 NarrowDependency。

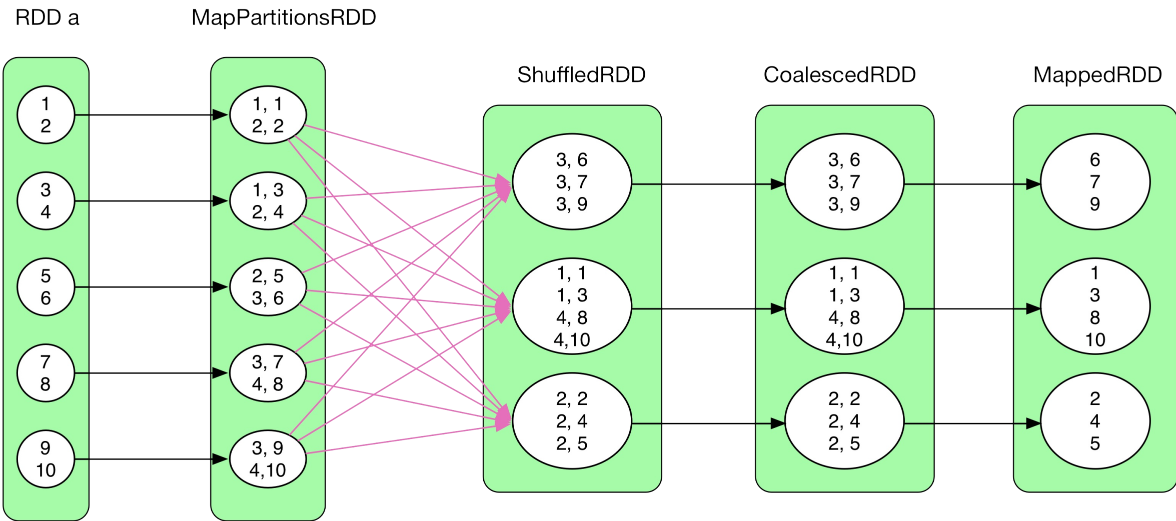
> CartesianRDD.getDependencies() 返回 rdds[RDD a, RDD b]。CartesianRDD 中的 partiton i 依赖于 (RDD a).List(i / numPartitionsInRDDb) 和 (RDD b).List(i % numPartitionsInRDDb)。

9) coalesce(numPartitions, shuffle = false)









coalesce() 可以将 parent RDD 的 partition 个数进行调整，比如从 5 个减少到 3 个，或者从 5 个增加到 10 个。需要注意的

是当 shuffle = false 的时候，是不能增加 partition 个数的（不能从 5 个变为 10 个）。

coalesce() 的核心问题是如何确立 **CoalescedRDD 中 partition 和其 parent RDD 中 partition** 的关系。

- coalesce(shuffle = false) 时，由于不能进行 shuffle，问题变为 **parent RDD** 中哪些**partition** 可以合并在一起。合并因素除了要考虑 partition 中元素个数外，还要考虑 locality 及 balance 的问题。因此，Spark 设计了一个非常复杂的算法来解决该问题（算法部分我还没有深究）。注意 `Example: a.coalesce(3, shuffle = false)` 展示了 N:1 的 NarrowDependency。
- coalesce(shuffle = true) 时，由于可以进行 **shuffle**，问题变为如何将 **RDD** 中所有 **records** 平均划分到 **N** 个 **partition** 中。很简单，在每个 partition 中，给每个 record 附加一个 key，key 递增，这样经过 hash(key) 后，key 可以被平均分配到不同的 partition 中，类似 Round-robin 算法。在第二个例子中，RDD a 中的每个元素，先被加上了递增的 key（如 MapPartitionsRDD 第二个 partition 中 (1, 3) 中的 1）。在每个 partition 中，第一个元素 (Key, Value) 中的 key 由 `(new Random(index)).nextInt(numPartitions)` 计算得到，index 是该 partition 的索引，numPartitions 是 CoalescedRDD 中的 partition 个数。接下来元素的 key 是递增的，然后 shuffle 后的 ShuffledRDD 可以得到均分的 records，然后经过复杂算法来建立 ShuffledRDD 和 CoalescedRDD 之间的数据联系，最后过滤掉 key，得到 coalesce 后的结果 MappedRDD。

10) repartition(numPartitions)

等价于 coalesce(numPartitions, shuffle = true)

Primitive transformation()

combineByKey()

分析了这么多 **RDD** 的逻辑执行图，它们之间有没有共同之处？如果有，是怎么被设计和实现的？

仔细分析 RDD 的逻辑执行图会发现，ShuffleDependency 左边的 RDD 中的 record 要求是 <key, value> 型的，经过 ShuffleDependency 后，包含相同 key 的 records 会被 aggregate 到一起，然后在 aggregated 的 records 上执行不同的计算逻辑。实际执行时（后面的章节会具体谈到）很多 transformation() 如 groupByKey()，reduceByKey() 是边 aggregate 数据边执行计算逻辑的，因此共同之处就是 **aggregate** 同时 **compute()**。Spark 使用 combineByKey() 来实现这个 aggregate + compute() 的基础操作。

combineByKey() 的定义如下：

```
def combineByKey[C](createCombiner: V => C,
      mergeValue: (C, V) => C,
      mergeCombiners: (C, C) => C,
      partitioner: Partitioner,
      mapSideCombine: Boolean = true,
      serializer: Serializer = null): RDD[(K, C)]
```

其中主要有三个参数 createCombiner，mergeValue 和 mergeCombiners。简单解释下这三个函数及 combineByKey() 的意义，注意它们的类型：

假设一组具有相同 K 的 <K, V> records 正在一个个流向 combineByKey()，createCombiner 将第一个 record 的 value 初始化为 c （比如，c = value），然后从第二个 record 开始，来一个 record 就使用 mergeValue(c, record.value) 来更新 c，比如想要对这些 records 的所有 values 做 sum，那么使用 c = c + record.value。等到 records 全部被 mergeValue()，得到结果 c。假设还有一组 records（key 与前面那组的 key 均相同）一个个到来，combineByKey() 使用前面的方法不断计算得到 c'。现在如果要求这两组 records 总的 combineByKey() 后的结果，那么可以使用 final c = mergeCombiners(c, c') 来计算。

Discussion

至此，我们讨论了如何生成 job 的逻辑执行图，这些图也是 Spark 看似简单的 API 背后的复杂计算逻辑及数据依赖关系。

整个 job 会产生哪些 RDD 由 transformation() 语义决定。一些 transformation()， 比如 cogroup() 会被很多其他操作用到。

RDD 本身的依赖关系由 transformation() 生成的每一个 RDD 本身语义决定。如 CoGroupedRDD 依赖于所有参加 cogroup() 的 RDDs。

RDD 中 partition 依赖关系分为 NarrowDependency 和 ShuffleDependency。前者是完全依赖，后者是部分依赖。NarrowDependency 里面又包含多种情况，只有前后两个 RDD 的 partition 个数以及 partitioner 都一样，才会出现 NarrowDependency。

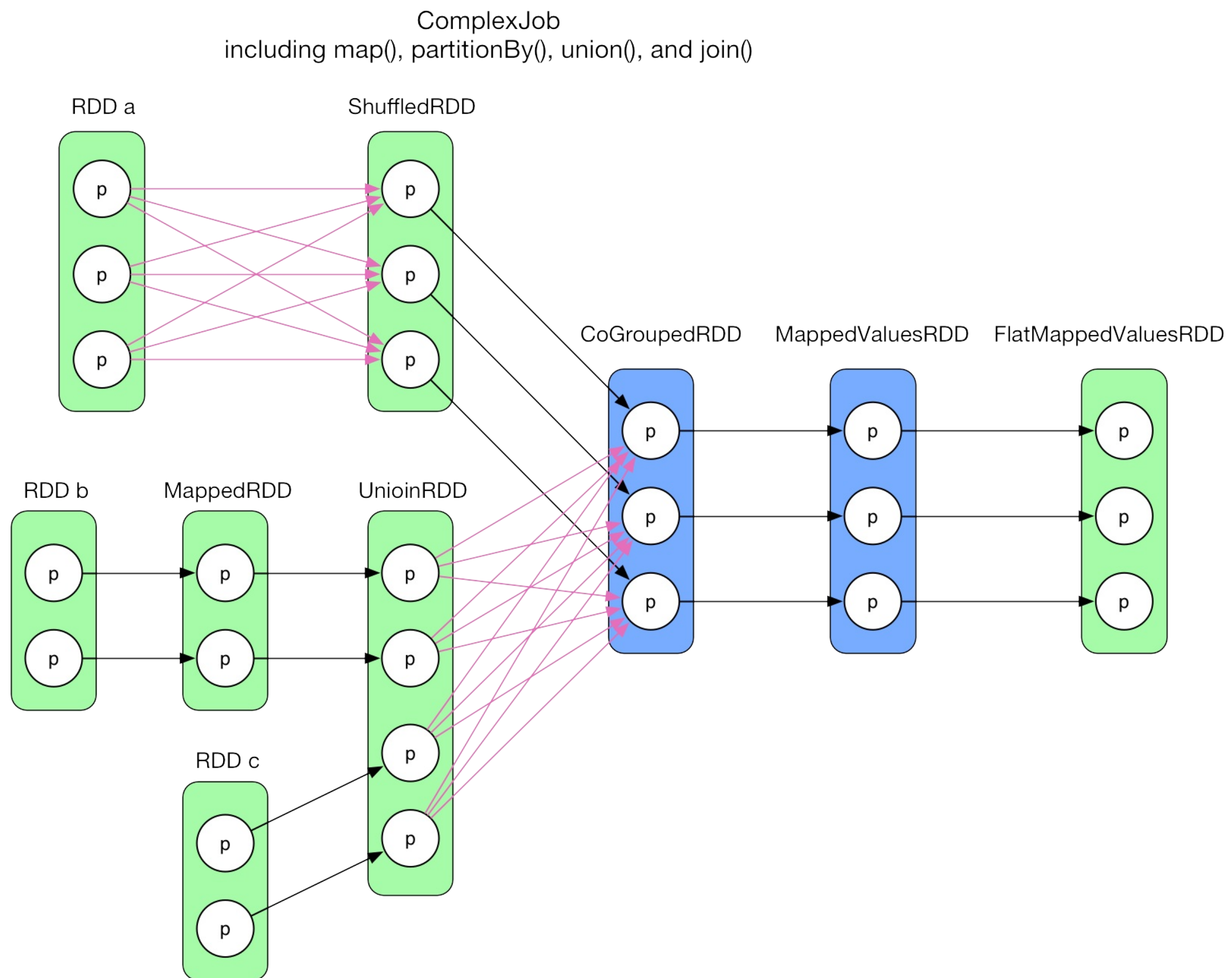
从数据处理逻辑的角度来看，MapReduce 相当于 Spark 中的 map() + reduceByKey()，但严格来讲 MapReduce 中的 reduce() 要比 reduceByKey() 的功能强大些，详细差别会在 Shuffle details 一章中继续讨论。

Job 物理执行图

在 Overview 里我们初步介绍了 DAG 型的物理执行图，里面包含 stages 和 tasks。这一章主要解决的问题是：

给定 **job** 的逻辑执行图，如何生成物理执行图（也就是 **stages** 和 **tasks**）？

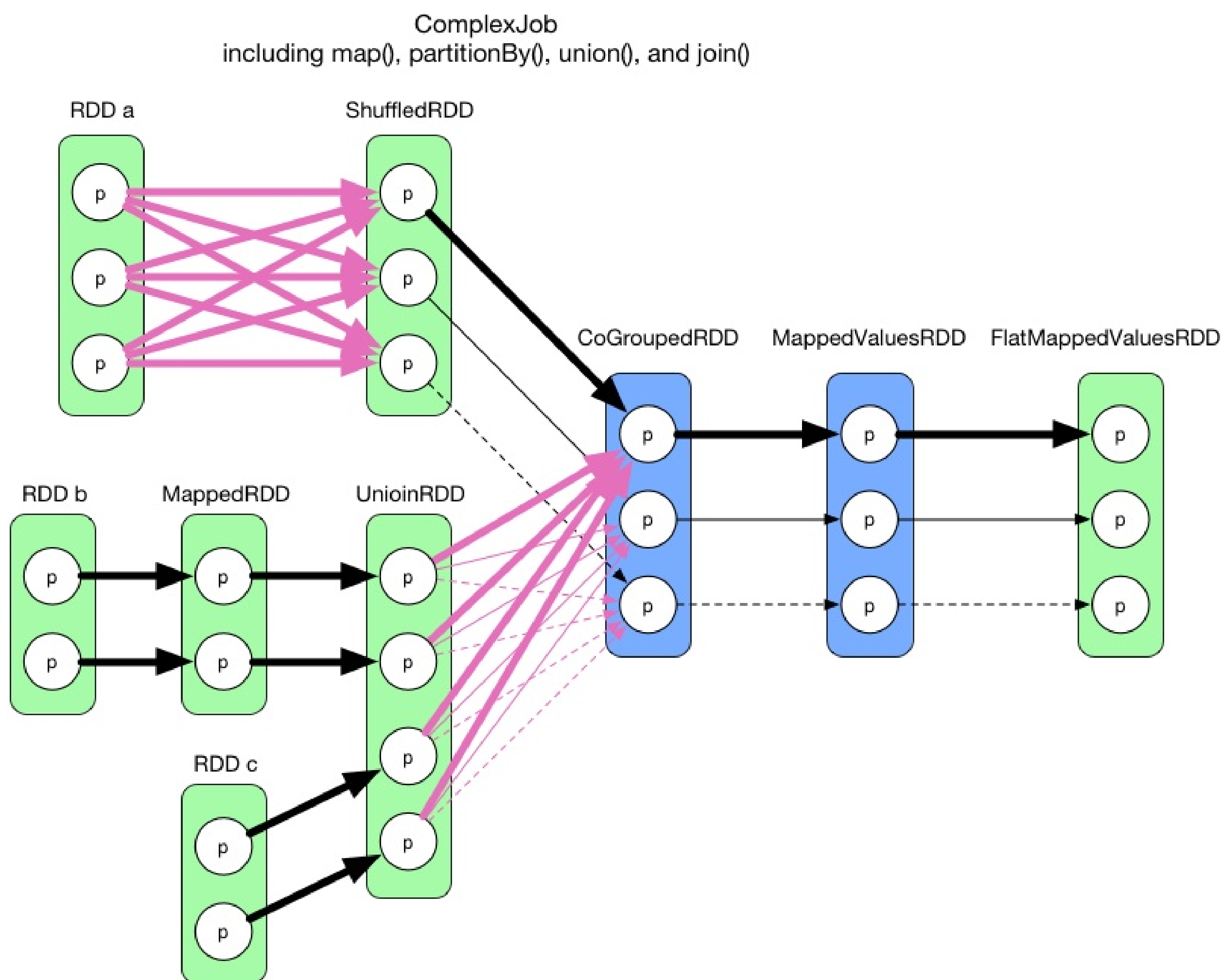
一个复杂 job 的逻辑执行图



代码贴在本章最后。给定这样一个复杂数据依赖图，如何合理划分 **stage**，并确定 **task** 的类型和个数？

一个直观想法是将前后关联的 RDDs 组成一个 stage，每个箭头生成一个 task。对于两个 RDD 聚合成一个 RDD 的情况，这三个 RDD 组成一个 stage。这样虽然可以解决问题，但显然效率不高。除了效率问题，这个想法还有一个更严重的问题：大量中间数据需要存储。对于 task 来说，其执行结果要么要存到磁盘，要么存到内存，或者两者皆有。如果每个箭头都是 task 的话，每个 RDD 里面的数据都需要存起来，占用空间可想而知。

仔细观察一下逻辑执行图会发现：在每个 RDD 中，每个 partition 是独立的，也就是说在 RDD 内部，每个 partition 的数据依赖各自不会相互干扰。因此，一个大胆的想法是将整个流程图看成一个 stage，为最后一个 finalRDD 中的每个 partition 分配一个 task。图示如下：



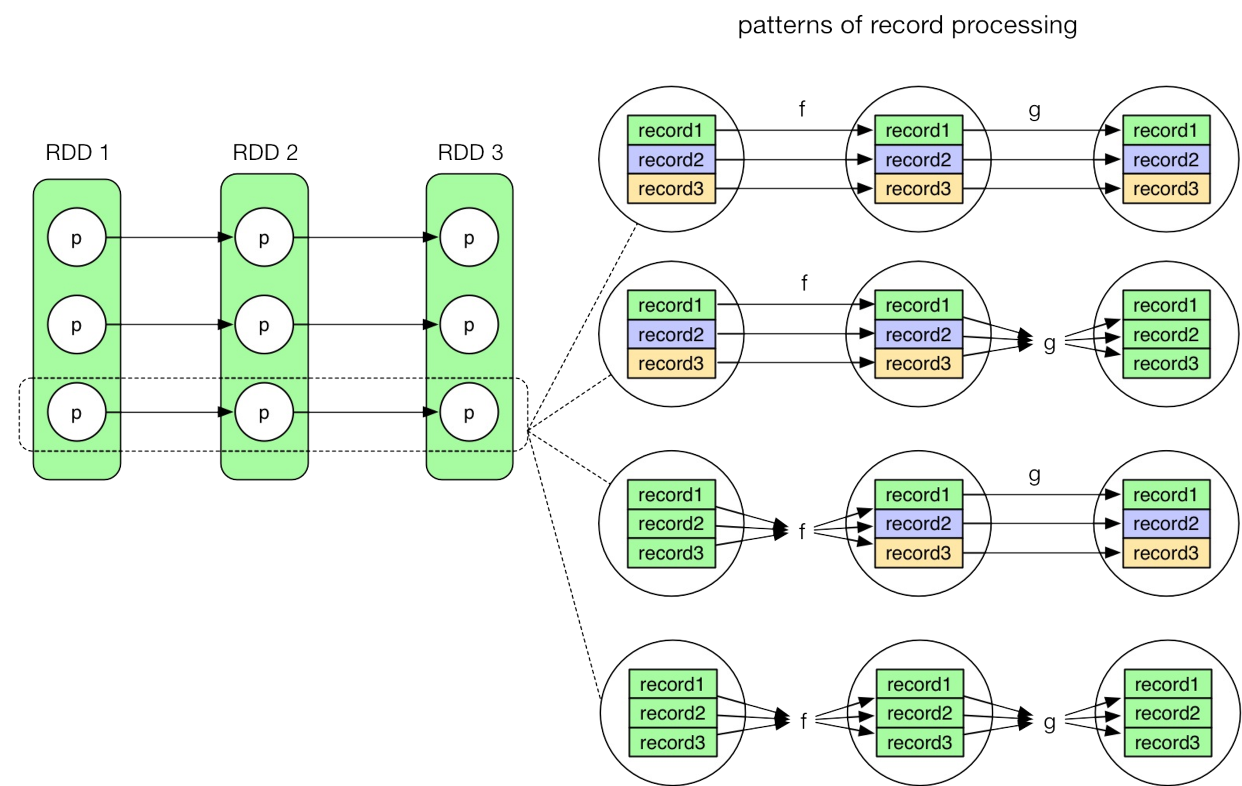
所有的粗箭头组合成第一个 task，该 task 计算结束后顺便将 CoGroupedRDD 中已经计算得到的第二个和第三个 partition 存起来。之后第二个 task（细实线）只需计算两步，第三个 task（细虚线）也只需要计算两步，最后得到结果。

这个想法有两个不靠谱的地方：

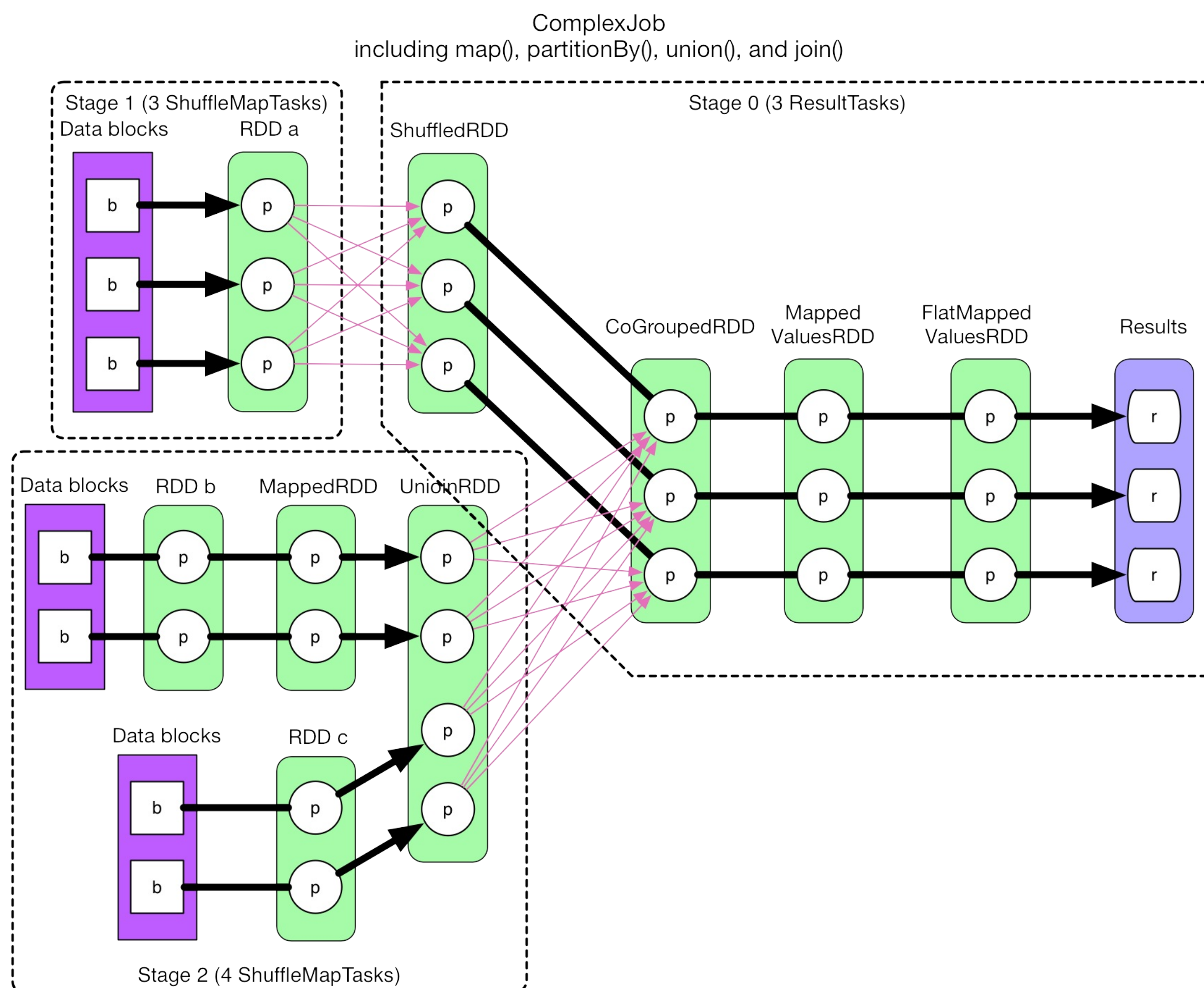
- 第一个 task 太大，碰到 ShuffleDependency 后，不得不计算 shuffle 依赖的 RDDs 的所有 partitions，而且都在这一个 task 里面计算。
- 需要设计巧妙的算法来判断哪个 RDD 中的哪些 partition 需要 cache。而且 cache 会占用存储空间。

虽然这是个不靠谱的想法，但有一个可取之处，即 **pipeline** 思想：数据用的时候再算，而且数据是流到要计算的位置的。比如在第一个 task 中，从 FlatMappedValuesRDD 中的 partition 向前推算，只计算要用的（依赖的） RDDs 及 partitions。在第二个 task 中，从 CoGroupedRDD 到 FlatMappedValuesRDD 计算过程中，不需要存储中间结果（MappedValuesRDD 中 partition 的全部数据）。

更进一步，从 record 粒度来讲，如下图中，第一个 pattern 中先算 g(f(record1))，然后原始的 record1 和 f(record1) 都可以丢掉，然后再算 g(f(record2))，丢掉中间结果，最后算 g(f(record3))。对于第二个 pattern 中的 g，record1 进入 g 后，理论上可以丢掉（除非被手动 cache）。其他 pattern 同理。



回到 stage 和 task 的划分问题，上面不靠谱想法的主要问题是碰到 ShuffleDependency 后无法进行 pipeline。那么只要在 ShuffleDependency 处断开，就只剩 NarrowDependency，而 NarrowDependency chain 是可以进行 pipeline 的。按照此思想，上面 ComplexJob 的划分图如下：

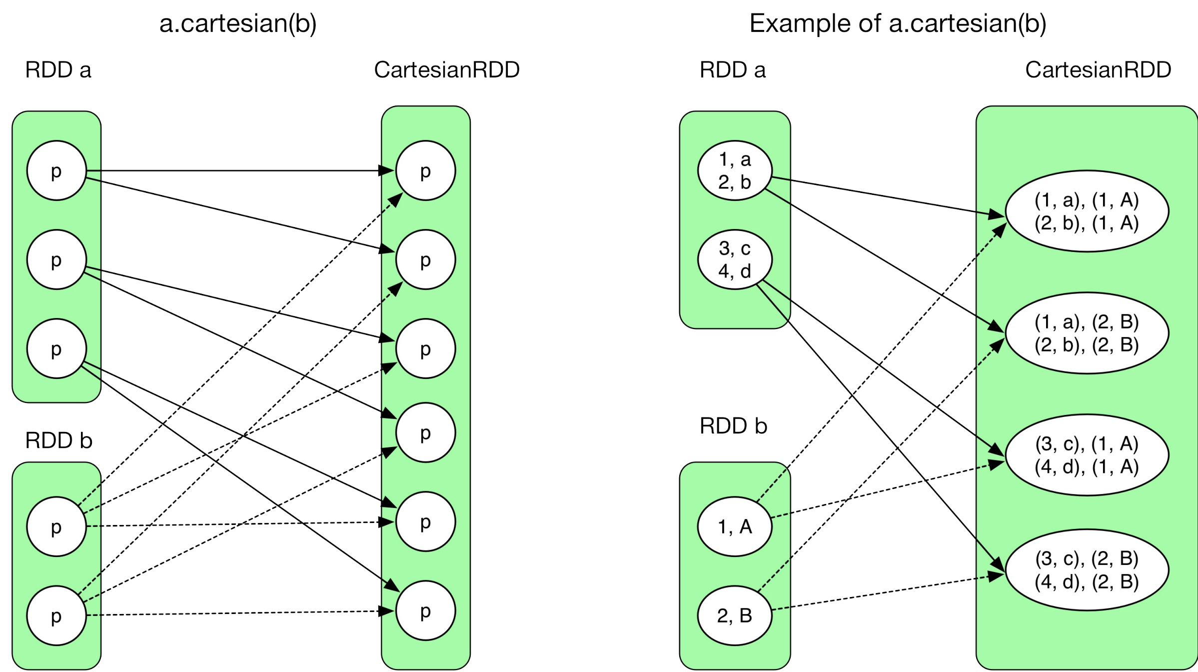


所以划分算法就是：从后往前推算，遇到 **ShuffleDependency** 就断开，遇到 **NarrowDependency** 就将其加入该 **stage**。每个 **stage** 里面 **task** 的数目由该 **stage** 最后一个 **RDD** 中的 **partition** 个数决定。

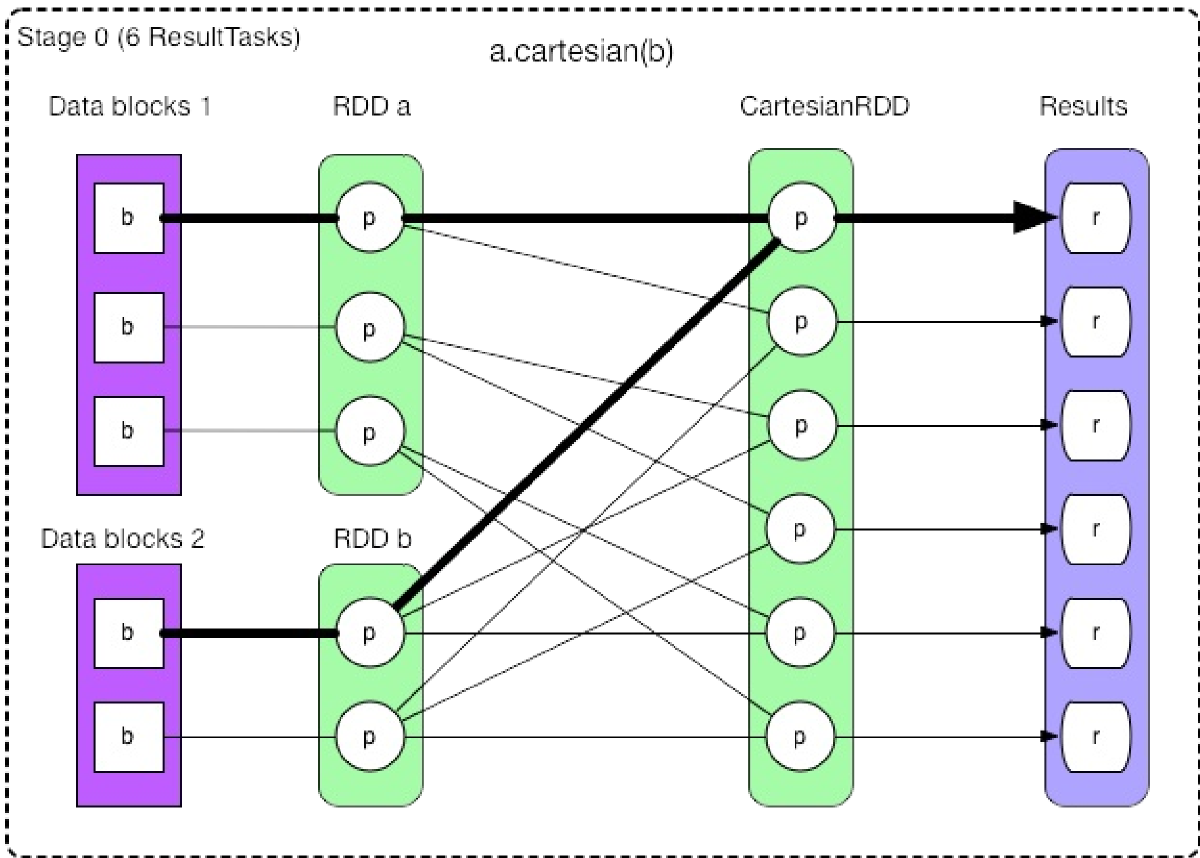
粗箭头表示 task。因为是从后往前推算，因此最后一个 stage 的 id 是 0，stage 1 和 stage 2 都是 stage 0 的 parents。如果 **stage** 最后要产生 **result**，那么该 **stage** 里面的 **task** 都是 **ResultTask**，否则都是 **ShuffleMapTask**。之所以称为 ShuffleMapTask 是因为其计算结果需要 shuffle 到下一个 stage，本质上相当于 MapReduce 中的 mapper。ResultTask 相当于 MapReduce 中的 reducer（如果需要从 parent stage 那里 shuffle 数据），也相当于普通 mapper（如果该 stage 没有 parent stage）。

还有一个问题：算法中提到 NarrowDependency chain 可以 pipeline，可是这里的 **ComplexJob** 只展示了 **OneToOneDependency** 和 **RangeDependency** 的 **pipeline**，普通 **NarrowDependency** 如何 **pipeline**？

回想上一章里面 cartesian(otherRDD) 里面复杂的 NarrowDependency，图示如下：



经过算法划分后结果如下：

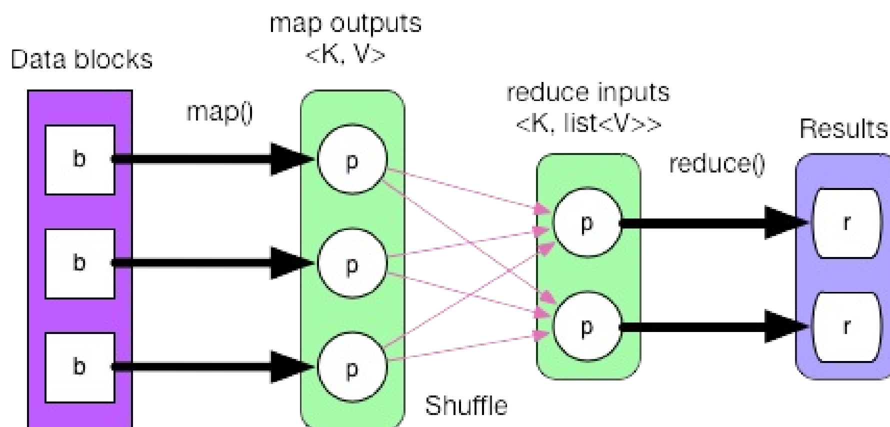


图中粗箭头展示了第一个 ResultTask，其他的 task 依此类推。由于该 stage 的 task 直接输出 result，所以这个图包含 6 个 ResultTasks。与 OneToOneDependency 不同的是这里每个 ResultTask 需要计算 3 个 RDD，读取两个 data block，而整个读取和计算这三个 RDD 的过程在一个 task 里面完成。当计算 CartesianRDD 中的 partition 时，需要从两个 RDD 获取 records，由于都在一个 task 里面，不需要 shuffle。这个图说明：不管是 **1:1** 还是 **N:1** 的 **NarrowDependency**，只要是 **NarrowDependency chain**，就可以进行 **pipeline**，生成的 **task** 个数与该 **stage** 最后一个 **RDD** 的 **partition** 个数相同。

物理图的执行

生成了 stage 和 task 以后，下一个问题就是 **task** 如何执行来生成最后的 **result**？

回到 ComplexJob 的物理执行图，如果按照 MapReduce 的逻辑，从前到后执行，map() 产生中间数据 map outpus，经过 partition 后放到本地磁盘。再经过 shuffle-sort-aggregate 后生成 reduce inputs，最后 reduce() 执行得到 result。执行流程如下：



整个执行流程没有问题，但不能直接套用在 Spark 的物理执行图上，因为 MapReduce 的流程图简单、固定，而且没有 pipeline。

回想 pipeline 的思想是 数据用的时候再算，而且数据是流到要计算的位置的。Result 产生的地方的就是要计算的位置，要确定 “需要计算的数据”，我们可以从后往前推，需要哪个 partition 就计算哪个 partition，如果 partition 里面没有数据，就继续向前推，形成 computing chain。这样推下去，结果就是：需要首先计算出每个 stage 最左边的 RDD 中的某些 partition。

对于没有 **parent stage** 的 **stage**，该 **stage** 最左边的 **RDD** 是可以立即计算的，而且每计算出一个 **record** 后便可以流入 **f** 或 **g**（见前面图中的 **patterns**）。如果 f 中的 record 关系是 1:1 的，那么 f(record1) 计算结果可以立即顺着 computing chain 流入 g 中。如果 f 的 record 关系是 N:1，record1 进入 f() 后也可以被回收。总结一下，computing chain 从后到前建立，而实际计算出的数据从前到后流动，而且计算出的第一个 record 流动到不能再流动后，再计算下一个 record。这样，虽然是要计算后续 RDD 的 partition 中的 records，但并不是要求当前 RDD 的 partition 中所有 records 计算得到后再整体向后流动。

对于有 parent stage 的 stage，先等着所有 parent stages 中 final RDD 中数据计算好，然后经过 shuffle 后，问题就又回到了计算 “没有 parent stage 的 stage”。

> 代码实现：每个 RDD 包含的 getDependency() 负责确立 RDD 的数据依赖，compute() 方法负责接收 parent RDDs 或者 data block 流入的 records，进行计算，然后输出 record。经常可以在 RDD 中看到这样的代码 `firstParent[T].iterator(split, context).map(f)`。firstParent 表示该 RDD 依赖的第一个 parent RDD，iterator() 表示 parentRDD 中的 records 是一个一个流入该 RDD 的，map(f) 表示每流入一个 recod 就对其进行 f(record) 操作，输出 record。为了统一接口，这段 compute() 仍然返回一个 iterator，来迭代 map(f) 输出的 records。

总结一下：整个 **computing chain** 根据数据依赖关系自后向前建立，遇到 **ShuffleDependency** 后形成 **stage**。在每个 **stage** 中，每个 **RDD** 中的 **compute()** 调用 **parentRDD.iter()** 来将 **parent RDDs** 中的 **records** 一个个 **fetch** 过来。

如果要自己设计一个 RDD，那么需要注意的是 compute() 只负责定义 parent RDDs => output records 的计算逻辑，具体依赖哪些 parent RDDs 由 `getDependency()` 定义，具体依赖 parent RDD 中的哪些 partitions 由 `dependency.getParents()` 定义。

例如，在 CartesianRDD 中，

```
  // RDD x = (RDD a).cartesian(RDD b)
  // 定义 RDD x 应该包含多少个 partition，每个 partition 是什么类型
  override def getPartitions: Array[Partition] = {
     // create the cross product split
     val array = new Array[Partition](rdd1.partitions.size * rdd2.partitions.size)
     for (s1 <- rdd1.partitions; s2 <- rdd2.partitions) {
       val idx = s1.index * numPartitionsInRdd2 + s2.index
       array(idx) = new CartesianPartition(idx, rdd1, rdd2, s1.index, s2.index)
     }
     array
   }

   // 定义 RDD x 中的每个 partition 怎么计算得到
   override def compute(split: Partition, context: TaskContext) = {
     val currSplit = split.asInstanceOf[CartesianPartition]
     // s1 表示 RDD x 中的 partition 依赖 RDD a 中的 partitions（这里只依赖一个）
     // s2 表示 RDD x 中的 partition 依赖 RDD b 中的 partitions（这里只依赖一个）
     for (x <- rdd1.iterator(currSplit.s1, context);
          y <- rdd2.iterator(currSplit.s2, context)) yield (x, y)
   }

   // 定义 RDD x 中的 partition i 依赖于哪些 RDD 中的哪些 partitions
   //
   // 这里 RDD x 依赖于 RDD a，同时依赖于 RDD b，都是 NarrowDependency
   // 对于第一个依赖，RDD x 中的 partition i 依赖于 RDD a 中的
   //    第 List(i / numPartitionsInRdd2) 个 partition
   // 对于第二个依赖，RDD x 中的 partition i 依赖于 RDD b 中的
   //    第 List(id % numPartitionsInRdd2) 个 partition
   override def getDependencies: Seq[Dependency[_]] = List(
     new NarrowDependency(rdd1) {
       def getParents(id: Int): Seq[Int] = List(id / numPartitionsInRdd2)
     },
     new NarrowDependency(rdd2) {
       def getParents(id: Int): Seq[Int] = List(id % numPartitionsInRdd2)
     }
   )
```

生成 job

前面介绍了逻辑和物理执行图的生成原理，那么，怎么触发 **job** 的生成？已经介绍了 **task**，那么 **job** 是什么？

下表列出了可以触发执行图生成的典型 action()，其中第二列是 `processPartition()`，定义如何计算 partition 中的 records 得到 result。第三列是 `resultHandler()`，定义如何对从各个 partition 收集来的 results 进行计算来得到最终结果。

Action	finalRDD(records) => result	compute(results)
reduce(func)	(record1, record2) => result, (result, record i) => result	(result1, result 2) => result, (result, result i) => result
collect()	Array[records] => result	Array[result]
count()	count(records) => result	sum(result)
foreach(f)	f(records) => result	Array[result]
take(n)	record (i<=n) => result	Array[result]
first()	record 1 => result	Array[result]
takeSample()	selected records => result	Array[result]
takeOrdered(n, [ordering])	TopN(records) => result	TopN(results)
saveAsHadoopFile(path)	records => write(records)	null
countByKey()	(K, V) => Map(K, count(K))	(Map, Map) => Map(K, count(K))

用户的 driver 程序中一旦出现 action()，就会生成一个 job，比如 `foreach()` 会调用 `sc.runJob(this, (iter: Iterator[T]) =>`

`iter.foreach(f))`，向 DAGScheduler 提交 job。如果 driver 程序后面还有 action()，那么其他 action() 也会生成 job 提交。所以，driver 有多少个 action()，就会生成多少个 job。这就是 Spark 称 driver 程序为 application（可能包含多个 job）而不是 job 的原因。

每一个 job 包含 n 个 stage，最后一个 stage 产生 result。比如，第一章的 GroupByTest 例子中存在两个 job，一共产生了两组 result。在提交 job 过程中，DAGScheduler 会首先划分 stage，然后先提交无 **parent stage** 的 **stages**，并在提交过程中确定该 stage 的 task 个数及类型，并提交具体的 task。无 parent stage 的 stage 提交完后，依赖该 stage 的 stage 才能够提交。从 stage 和 task 的执行角度来讲，一个 stage 的 parent stages 执行完后，该 stage 才能执行。

提交 job 的实现细节

下面简单分析下 job 的生成和提交代码，提交过程在 Architecture 那一章也会有图文并茂的分析：

1. rdd.action() 会调用 `DAGScheduler.runJob(rdd, processPartition, resultHandler)` 来生成 job。
2. runJob() 会首先通过 `rdd.getPartitions()` 来得到 finalRDD 中应该存在的 partition 的个数和类型：Array[Partition]。然后根据 partition 个数 new 出来将来要持有 result 的数组 `Array[Result](partitions.size)`。
3. 最后调用 DAGScheduler 的 `runJob(rdd, cleanedFunc, partitions, allowLocal, resultHandler)` 来提交 job。cleanedFunc 是 processParittion 经过闭包清理后的结果，这样可以被序列化后传递给不同节点的 task。
4. DAGScheduler 的 runJob 继续调用 `submitJob(rdd, func, partitions, allowLocal, resultHandler)` 来提交 job。
5. submitJob() 首先得到一个 jobId，然后再次包装 func，向 DAGSchedulerEventProcessActor 发送 JobSubmitted 信息，该 actor 收到信息后进一步调用 `dagScheduler.handleJobSubmitted()` 来处理提交的 job。之所以这么麻烦，是为了符合事件驱动模型。
6. handleJobSubmmitted() 首先调用 finalStage = newStage() 来划分 stage，然后submitStage(finalStage)。由于 finalStage 可能有 parent stages，实际先提交 parent stages，等到他们执行完，finalStage 需要再次提交执行。再次提交由 handleJobSubmmitted() 最后的 submitWaitingStages() 负责。

分析一下 newStage() 如何划分 stage：

1. 该方法在 new Stage() 的时候会调用 finalRDD 的 getParentStages()。
2. getParentStages() 从 finalRDD 出发，反向 visit 逻辑执行图，遇到 NarrowDependency 就将依赖的 RDD 加入到 stage，遇到 ShuffleDependency 切开 stage，并递归到 ShuffleDepedency 依赖的 stage。
3. 一个 ShuffleMapStage（不是最后形成 result 的 stage）形成后，会将该 stage 最后一个 RDD 注册到 `MapOutputTrackerMaster.registerShuffle(shuffleDep.shuffleId, rdd.partitions.size)`，这一步很重要，因为 shuffle 过程需要 MapOutputTrackerMaster 来指示 ShuffleMapTask 输出数据的位置。

分析一下 submitStage(stage) 如何提交 stage 和 task：

1. 先确定该 stage 的 missingParentStages，使用 `getMissingParentStages(stage)`。如果 parentStages 都可能已经执行过了，那么就为空了。
2. 如果 missingParentStages 不为空，那么先递归提交 missing 的 parent stages，并将自己加入到 waitingStages 里面，等到 parent stages 执行结束后，会触发提交 waitingStages 里面的 stage。
3. 如果 missingParentStages 为空，说明该 stage 可以立即执行，那么就调用 `submitMissingTasks(stage, jobId)` 来生成和提交具体的 task。如果 stage 是 ShuffleMapStage，那么 new 出来与该 stage 最后一个 RDD 的 partition 数相同的 ShuffleMapTasks。如果 stage 是 ResultStage，那么 new 出来与 stage 最后一个 RDD 的 partition 个数相同的 ResultTasks。一个 stage 里面的 task 组成一个 TaskSet，最后调用 `taskScheduler.submitTasks(taskSet)` 来提交一整个 taskSet。
4. 这个 taskScheduler 类型是 TaskSchedulerImpl，在 submitTasks() 里面，每一个 taskSet 被包装成 manager: TaskSetMananger，然后交给 `schedulableBuilder.addTaskSetManager(manager)`。schedulableBuilder 可以是 FIFOSchedulableBuilder 或者 FairSchedulableBuilder 调度器。submitTasks() 最后一步是通知 `backend.reviveOffers()` 去执行 task，backend 的类型是 SchedulerBackend。如果在集群上运行，那么这个 backend 类型是 SparkDeploySchedulerBackend。
5. SparkDeploySchedulerBackend 是 CoarseGrainedSchedulerBackend 的子类，`backend.reviveOffers()` 其实是向 DriverActor 发送 ReviveOffers 信息。SparkDeploySchedulerBackend 在 start() 的时候，会启动 DriverActor。

DriverActor 收到 ReviveOffers 消息后，会调用 `launchTasks(scheduler.resourceOffers(Seq(new WorkerOffer(executorId, executorHost(executorId), freeCores(executorId)))))` 来 launch tasks。scheduler 就是 TaskSchedulerImpl。 `scheduler.resourceOffers()` 从 FIFO 或者 Fair 调度器那里获得排序后的 TaskSetManager，并经过 `TaskSchedulerImpl.resourceOffer()`，考虑 locality 等因素来确定 task 的全部信息 TaskDescription。调度细节这里暂不讨论。

6. DriverActor 中的 launchTasks() 将每个 task 序列化，如果序列化大小不超过 Akka 的 akkaFrameSize，那么直接将 task 送到 executor 那里执行 `executorActor(task.executorId) ! LaunchTask(new SerializableBuffer(serializedTask))`。

Discussion

至此，我们讨论了：

- driver 程序如何触发 job 的生成
- 如何从逻辑执行图得到物理执行图
- pipeline 思想与实现
- 生成与提交 job 的实际代码

还有很多地方没有深入讨论，如：

- 连接 stage 的 shuffle 过程
- task 运行过程及运行位置

下一章重点讨论 shuffle 过程。

从逻辑执行图的建立，到将其转换成物理执行图的过程很经典，过程中的 dependency 划分，pipeline，stage 分割，task 生成 都是有条不紊，有理有据的。

ComplexJob 的源代码

```
package internals

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.HashPartitioner


object complexJob {
  def main(args: Array[String]) {

    val sc = new SparkContext("local", "ComplexJob test")

    val data1 = Array[(Int, Char)](
      (1, 'a'), (2, 'b'),
      (3, 'c'), (4, 'd'),
      (5, 'e'), (3, 'f'),
      (2, 'g'), (1, 'h'))
    val rangePairs1 = sc.parallelize(data1, 3)

    val hashPairs1 = rangePairs1.partitionBy(new HashPartitioner(3))


    val data2 = Array[(Int, String)]((1, "A"), (2, "B"),
      (3, "C"), (4, "D"))

    val pairs2 = sc.parallelize(data2, 2)
    val rangePairs2 = pairs2.map(x => (x._1, x._2.charAt(0)))


    val data3 = Array[(Int, Char)]((1, 'X'), (2, 'Y'))
    val rangePairs3 = sc.parallelize(data3, 2)
```

```
    val rangePairs = rangePairs2.union(rangePairs3)


    val result = hashPairs1.join(rangePairs)

    result.foreachWith(i => i)((x, i) => println("[result " + i + "] " + x))

    println(result.toDebugString)
  }
}
```

Shuffle 过程

上一章里讨论了 job 的物理执行图，也讨论了流入 RDD 中的 records 是怎么被 compute() 后流到后续 RDD 的，同时也分析了 task 是怎么产生 result，以及 result 怎么被收集后计算出最终结果的。然而，我们还没有讨论数据是怎么通过 **ShuffleDependency** 流向下一个 **stage** 的？

对比 Hadoop MapReduce 和 Spark 的 Shuffle 过程

如果熟悉 Hadoop MapReduce 中的 shuffle 过程，可能会按照 MapReduce 的思路去想象 Spark 的 shuffle 过程。然而，它们之间有一些区别和联系。

从 **high-level** 的角度来看，两者并没有大的差别。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce() （Spark 里可能是后续的一系列操作）。

从 **low-level** 的角度来看，两者差别不小。 Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。这样的好处在于 combine/reduce() 可以处理大规模的数据，因为其输入数据可以通过外排得到（mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并）。目前的 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey() 的操作；如果你是Spark 1.1的用户，可以将spark.shuffle.manager设置为sort，则会对数据进行排序。在Spark 1.2中，sort将作为默认的Shuffle实现。

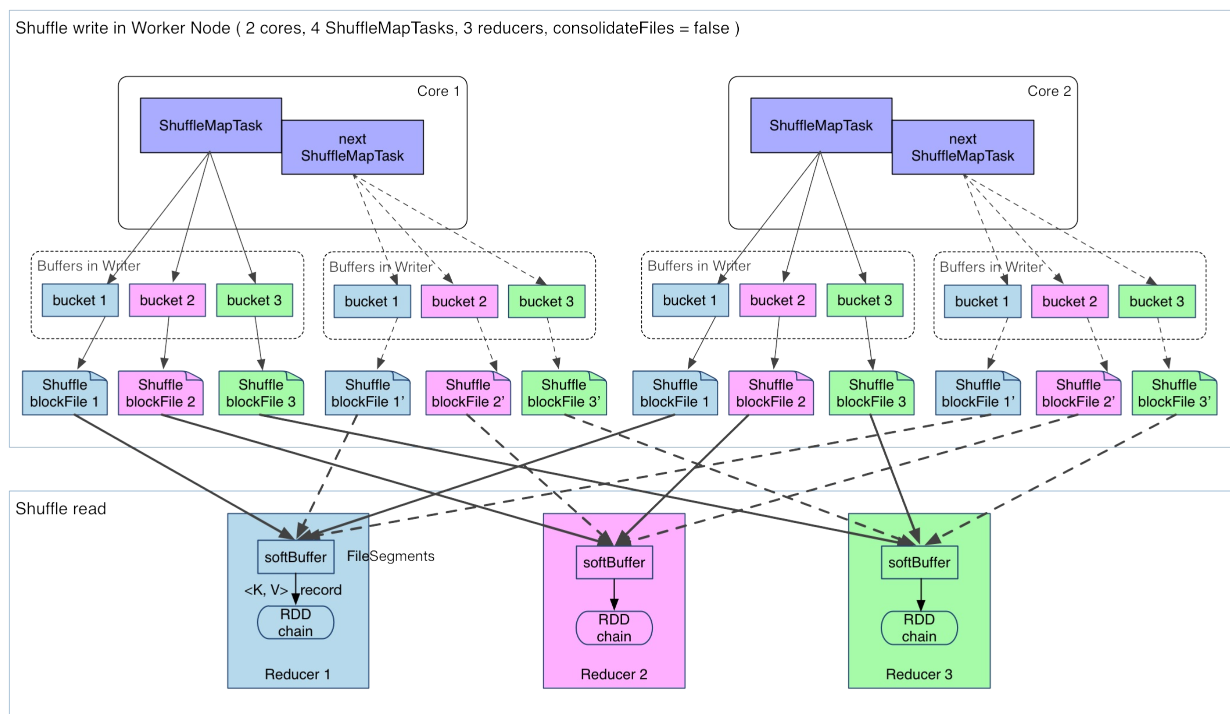
从实现角度来看，两者也有不少差别。 Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spill, merge, shuffle, sort, reduce() 等。每个阶段各司其职，可以按照过程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，所以 spill, merge, aggregate 等操作需要蕴含在 transformation() 中。

如果我们将 map 端划分数据、持久化数据的过程称为 shuffle write，而将 reducer 读入数据、aggregate 数据的过程称为 shuffle read。那么在 Spark 中，问题就变为怎么在 **job** 的逻辑或者物理执行图中加入 **shuffle write** 和 **shuffle read** 的处理逻辑？以及两个处理逻辑应该怎么高效实现？

Shuffle write

由于不要求数据有序，shuffle write 的任务很简单：将数据 partition 好，并持久化。之所以要持久化，一方面是要减少内存存储空间压力，另一方面也是为了 fault-tolerance。

shuffle write 的任务很简单，那么实现也很简单：将 shuffle write 的处理逻辑加入到 ShuffleMapStage（ShuffleMapTask 所在的 stage） 的最后，该 stage 的 final RDD 每输出一个 record 就将其 partition 并持久化。图示如下：



上图有 4 个 ShuffleMapTask 要在同一个 worker node 上运行，CPU core 数为 2，可以同时运行两个 task。每个 task 的执行结果（该 stage 的 finalRDD 中某个 partition 包含的 records）被逐一写到本地磁盘上。每个 task 包含 R 个缓冲区，R = reducer 个数（也就是下一个 stage 中 task 的个数），缓冲区被称为 bucket，其大小为 `spark.shuffle.file.buffer.kb` ，默认是 32KB（Spark 1.1 版本以前是 100KB）。

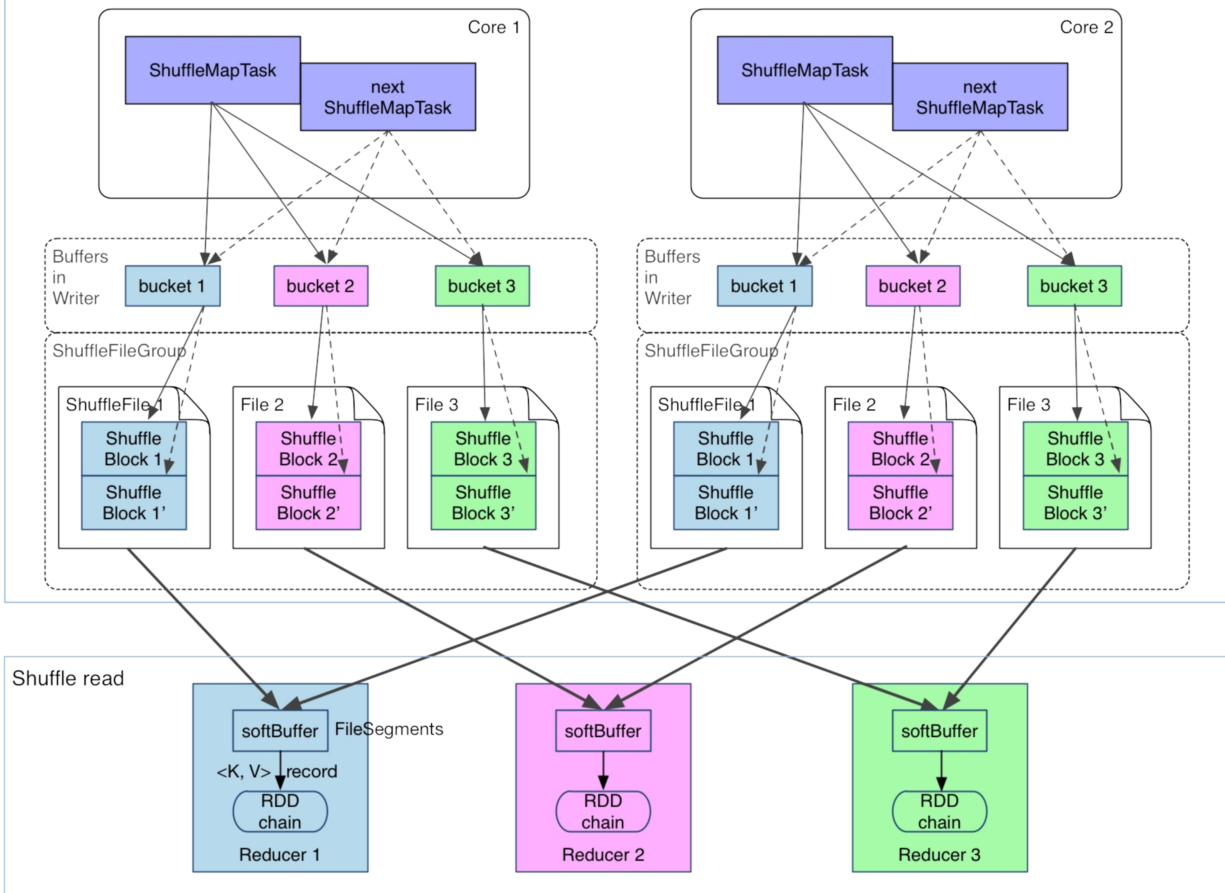
> 其实 bucket 是一个广义的概念，代表 ShuffleMapTask 输出结果经过 partition 后要存放的地方，这里为了细化数据存放位置和数据名称，仅仅用 bucket 表示缓冲区。

ShuffleMapTask 的执行过程很简单：先利用 pipeline 计算得到 finalRDD 中对应 partition 的 records。每得到一个 record 就将其送到对应的 bucket 里，具体是哪个 bucket 由 `partitioner.partition(record.getKey()))` 决定。每个 bucket 里面的数据会不断被写到本地磁盘上，形成一个 ShuffleBlockFile，或者简称 **FileSegment**。之后的 reducer 会去 fetch 属于自己的 FileSegment，进入 shuffle read 阶段。

这样的实现很简单，但有几个问题：

1. 产生的 **FileSegment** 过多。每个 ShuffleMapTask 产生 R（reducer 个数）个 FileSegment，M 个 ShuffleMapTask 就会产生 M * R 个文件。一般 Spark job 的 M 和 R 都很大，因此磁盘上会存在大量的数据文件。
2. 缓冲区占用内存空间大。每个 ShuffleMapTask 需要开 R 个 bucket，M 个 ShuffleMapTask 就会产生 M *R 个 bucket。虽然一个 ShuffleMapTask 结束后，对应的缓冲区可以被回收，但一个 worker node 上同时存在的 bucket 个数可以达到 cores* R 个（一般 worker 同时可以运行 cores 个 ShuffleMapTask），占用的内存空间也就达到了 `cores * R * 32 KB` 。对于 8 核 1000 个 reducer 来说，占用内存就是 256MB。

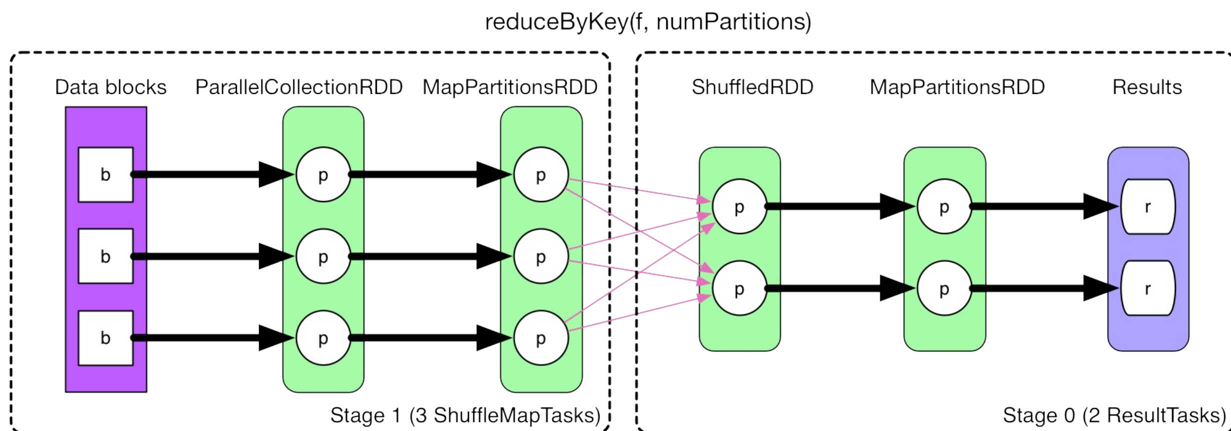
目前来看，第二个问题还没有好的方法解决，因为写磁盘终究是要开缓冲区的，缓冲区太小会影响 IO 速度。但第一个问题有一些方法去解决，下面介绍已经在 Spark 里面实现的 FileConsolidation 方法。先上图：

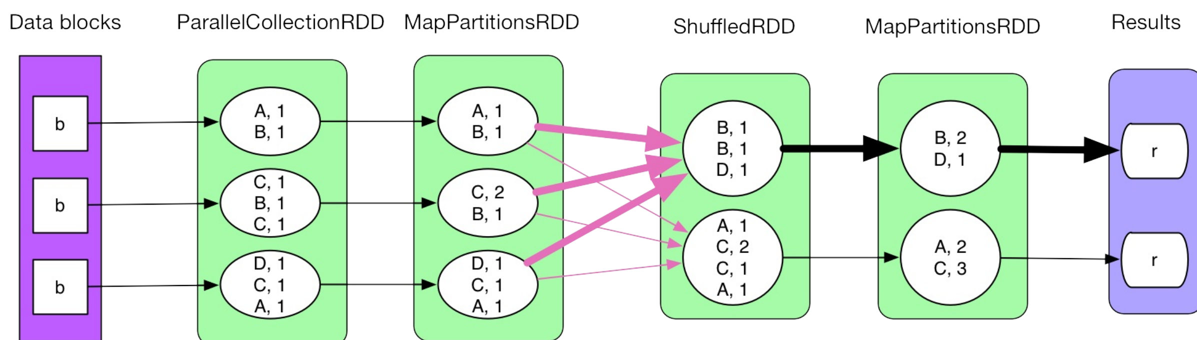


可以明显看出，在一个 core 上连续执行的 ShuffleMapTasks 可以共用一个输出文件 ShuffleFile。先执行完的 ShuffleMapTask 形成 ShuffleBlock i，后执行的 ShuffleMapTask 可以将输出数据直接追加到 ShuffleBlock i 后面，形成 ShuffleBlock i'，每个 ShuffleBlock 被称为 **FileSegment**。下一个 stage 的 reducer 只需要 fetch 整个 ShuffleFile 就行了。这样，每个 worker 持有的文件数降为 cores * R。FileConsolidation 功能可以通过 `spark.shuffle.consolidateFiles=true` 来开启。

Shuffle read

先看一张包含 ShuffleDependency 的物理执行图，来自 reduceByKey：





很自然地，要计算 ShuffleRDD 中的数据，必须先把 MapPartitionsRDD 中的数据 fetch 过来。那么问题就来了：

- 在什么时候 fetch，parent stage 中的一个 ShuffleMapTask 执行完还是等全部 ShuffleMapTasks 执行完？
- 边 fetch 边处理还是一次性 fetch 完再处理？
- fetch 来的数据存放到哪里？
- 怎么获得要 fetch 的数据的存放位置？

解决问题：

- 在什么时候 **fetch**？当 parent stage 的所有 ShuffleMapTasks 结束后再 fetch。理论上讲，一个 ShuffleMapTask 结束后就可以 fetch，但是为了迎合 stage 的概念（即一个 stage 如果其 parent stages 没有执行完，自己是不能被提交执行的），还是选择全部 ShuffleMapTasks 执行完再去 fetch。因为 fetch 来的 FileSegments 要先在内存做缓冲，所以一次 fetch 的 FileSegments 总大小不能太大。Spark 规定这个缓冲界限不能超过 `spark.reducer.maxMbInFlight`，这里用 **softBuffer** 表示，默认大小为 48MB。一个 softBuffer 里面一般包含多个 FileSegment，但如果某个 FileSegment 特别大的话，这一个就可以填满甚至超过 softBuffer 的界限。
- 边 **fetch** 边处理还是一次性 **fetch** 完再处理？边 fetch 边处理。本质上，MapReduce shuffle 阶段就是边 fetch 边使用 combine() 进行处理，只是 combine() 处理的是部分数据。MapReduce 为了让进入 reduce() 的 records 有序，必须等到全部数据都 shuffle-sort 后再开始 reduce()。因为 Spark 不要求 shuffle 后的数据全局有序，因此没必要等到全部数据 shuffle 完成后再处理。那么如何实现边 **shuffle** 边处理，而且流入的 **records** 是无序的？答案是使用可以 aggregate 的数据结构，比如 HashMap。每 shuffle 得到（从缓冲的 FileSegment 中 deserialize 出来）一个 \ record，直接将其放进 HashMap 里面。如果该 HashMap 已经存在相应的 Key，那么直接进行 aggregate 也就是 `func(hashMap.get(Key), Value)`，比如上面 WordCount 例子中的 func 就是 `hashMap.get(Key) + Value`，并将 func 的结果重新 put(key) 到 HashMap 中去。这个 func 功能上相当于 reduce()，但实际处理数据的方式与 MapReduce reduce() 有差别，差别相当于下面两段程序的差别。

```
// MapReduce
reduce(K key, Iterable<V> values) {
    result = process(key, values)
```

```
        return result
    }

    // Spark
    reduce(K key, Iterable<V> values) {
        result = null
        for (V value : values)
            result  = func(result, value)
        return result
    }
```

MapReduce 可以在 process 函数里面可以定义任何数据结构，也可以将部分或全部的 values 都 cache 后再进行处理，非常灵活。而 Spark 中的 func 的输入参数是固定的，一个是上一个 record 的处理结果，另一个是当前读入的 record，它们经过 func 处理后的结果被下一个 record 处理时使用。因此一些算法比如求平均数，在 process 里面很好实现，直接 `sum(values)/values.length`，而在 Spark 中 func 可以实现 `sum(values)`，但不好实现 `/values.length`。更多的 func 将会在下面的章节细致分析。

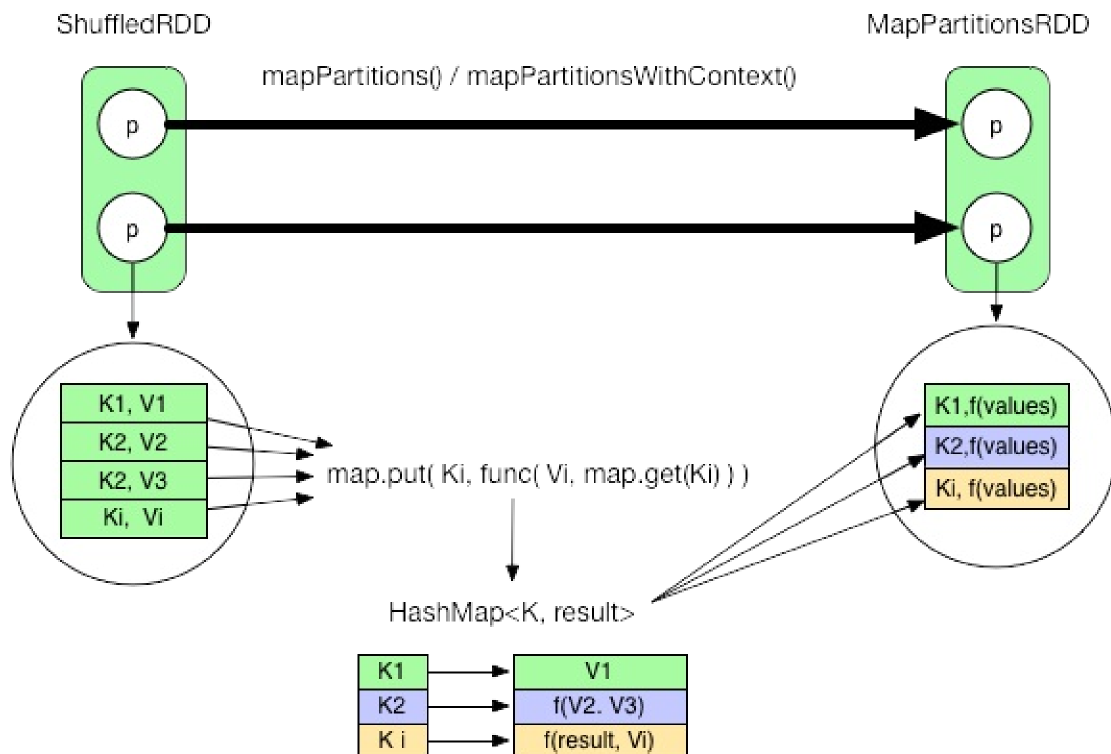
- **fetch** 来的数据存放到哪里？刚 fetch 来的 FileSegment 存放在 softBuffer 缓冲区，经过处理后的数据放在内存 + 磁盘上。这里我们主要讨论处理后的数据，可以灵活设置这些数据是“只用内存”还是“内存＋磁盘”。如果 `spark.shuffle.spill = false` 就只用内存。内存使用的是 `AppendOnlyMap`，类似 Java 的 `HashMap`，内存＋磁盘使用的是 `ExternalAppendOnlyMap`，如果内存空间不足时，`ExternalAppendOnlyMap` 可以将 \ records 进行 sort 后 spill 到磁盘上，等到需要它们的时候再进行归并，后面会详解。使用“内存＋磁盘”的一个主要问题就是如何在两者之间取得平衡？在 Hadoop MapReduce 中，默认将 reducer 的 70% 的内存空间用于存放 shuffle 来的数据，等到这个空间利用率达到 66% 的时候就开始 merge-combine()-spill。在 Spark 中，也适用同样的策略，一旦 ExternalAppendOnlyMap 达到一个阈值就开始 spill，具体细节下面会讨论。
- 怎么获得要 **fetch** 的数据的存放位置？在上一章讨论物理执行图中的 stage 划分的时候，我们强调 “一个 ShuffleMapStage 形成后，会将该 stage 最后一个 final RDD 注册到 `MapOutputTrackerMaster.registerShuffle(shuffleId, rdd.partitions.size)`，这一步很重要，因为 shuffle 过程需要 MapOutputTrackerMaster 来指示 ShuffleMapTask 输出数据的位置”。因此，reducer 在 shuffle 的时候是要去 driver 里面的 MapOutputTrackerMaster 询问 ShuffleMapTask 输出的数据位置的。每个 ShuffleMapTask 完成时会将 FileSegment 的存储位置信息汇报给 MapOutputTrackerMaster。

至此，我们已经讨论了 shuffle write 和 shuffle read 设计的核心思想、算法及某些实现。接下来，我们深入一些细节来讨论。

典型 transformation() 的 shuffle read

1. reduceByKey(func)

上面初步介绍了 reduceByKey() 是如何实现边 fetch 边 reduce() 的。需要注意的是虽然 Example(WordCount) 中给出了各个 RDD 的内容，但一个 partition 里面的 records 并不是同时存在的。比如在 ShuffledRDD 中，每 fetch 来一个 record 就立即进入了 func 进行处理。MapPartitionsRDD 中的数据是 func 在全部 records 上的处理结果。从 record 粒度上来看，reduce() 可以表示如下：



可以看到，fetch 来的 records 被逐个 aggreagte 到 HashMap 中，等到所有 records 都进入 HashMap，就得到最后的处理结果。唯一要求是 func 必须是 commulative 的（参见上面的 Spark 的 reduce() 的代码）。

ShuffledRDD 到 MapPartitionsRDD 使用的是 mapPartitionsWithContext 操作。

为了减少数据传输量，MapReduce 可以在 map 端先进行 combine()，其实在 Spark 也可以实现，只需要将上图 ShuffledRDD => MapPartitionsRDD 的 mapPartitionsWithContext 在 ShuffleMapStage 中也进行一次即可，比如 reduceByKey 例子中 ParallelCollectionRDD => MapPartitionsRDD 完成的就是 map 端的 combine()。

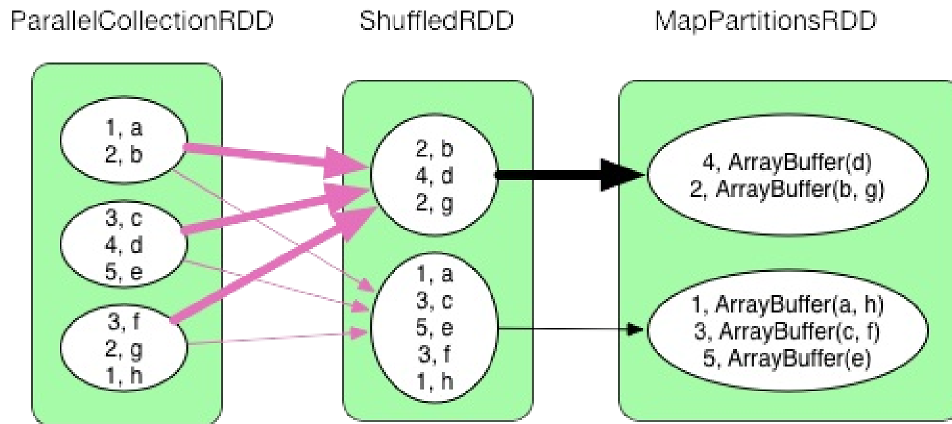
对比 **MapReduce** 的 **map()-reduce()** 和 **Spark** 中的 **reduceByKey()**：

- map 端的区别：map() 没有区别。对于 combine()，MapReduce 先 sort 再 combine()，Spark 直接在 HashMap 上进行 combine()。
- reduce 端区别：MapReduce 的 shuffle 阶段先 fetch 数据，数据量到达一定规模后 combine()，再将剩余数据 merge-sort 后 reduce()，reduce() 非常灵活。Spark 边 fetch 边 reduce()（在 HashMap 上执行 func），因此要求 func 符合 commulative 的特性。

从内存利用上来对比：

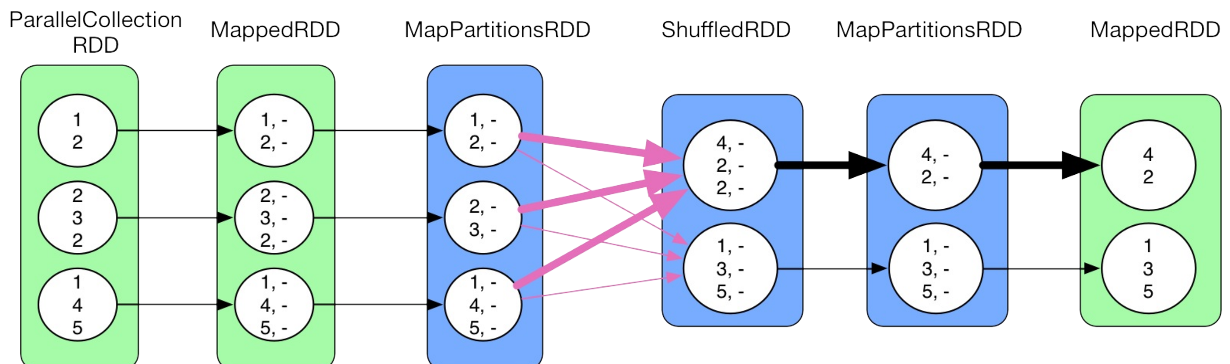
- map 端区别：MapReduce 需要开一个大型环形缓冲区来暂存和排序 map() 的部分输出结果，但 combine() 不需要额外空间（除非用户自己定义）。 Spark 需要 HashMap 内存数据结构来进行 combine()，同时输出 records 到磁盘上时也需要一个小的 buffer（bucket）。
- reduce 端区别：MapReduce 需要一部分内存空间来存储 shuffle 过来的数据，combine() 和 reduce() 不需要额外空间，因为它们的输入数据分段有序，只需归并一下就可以得到。在 Spark 中，fetch 时需要 softBuffer，处理数据时如果只使用内存，那么需要 HashMap 来持有处理后的结果。如果使用内存＋磁盘，那么在 HashMap 存放一部分处理后的数据。

2. groupByKey(numPartitions)



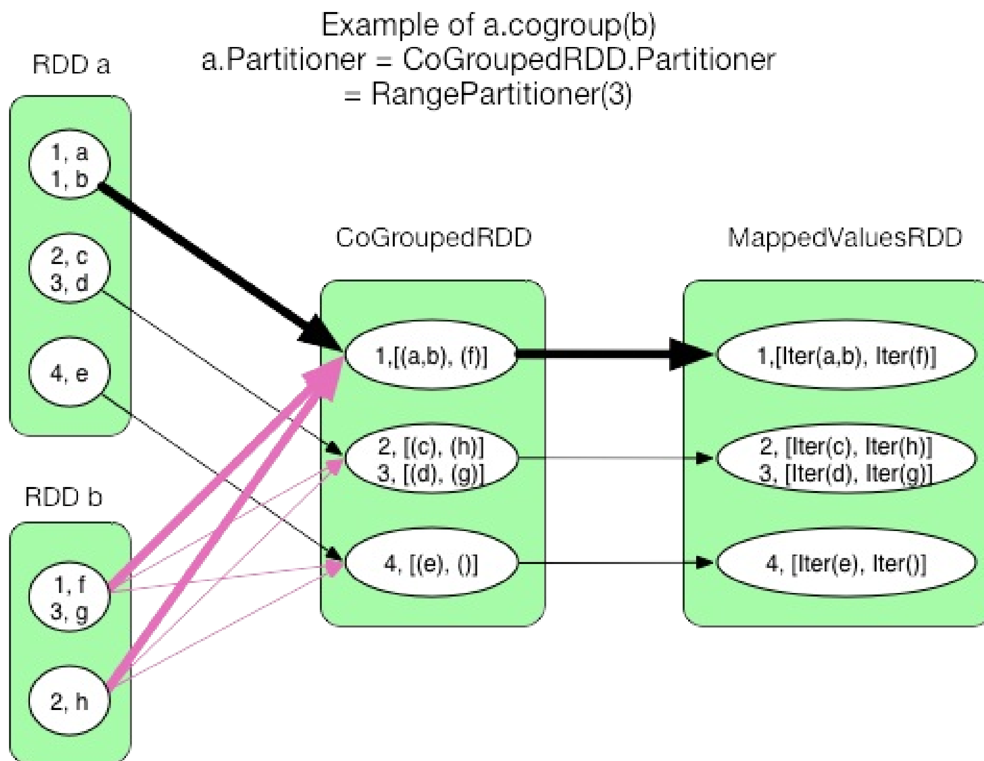
与 reduceByKey() 流程一样，只是 func 变成 `result = result ++ record.value`，功能是将每个 key 对应的所有 values 链接在一起。result 来自 hashMap.get(record.key)，计算后的 result 会再次被 put 到 hashMap 中。与 reduceByKey() 的区别就是 groupByKey() 没有 map 端的 combine()。对于 groupByKey() 来说 map 端的 combine() 只是减少了重复 Key 占用的空间，如果 key 重复率不高，没必要 combine()，否则，最好能够 combine()。

3. distinct(numPartitions)



与 reduceByKey() 流程一样，只是 func 变成 `result = result == null? record.value : result`，如果 HashMap 中没有该 record 就将其放入，否则舍弃。与 reduceByKey() 相同，在map 端存在 combine()。

4. cogroup(otherRDD, numPartitions)

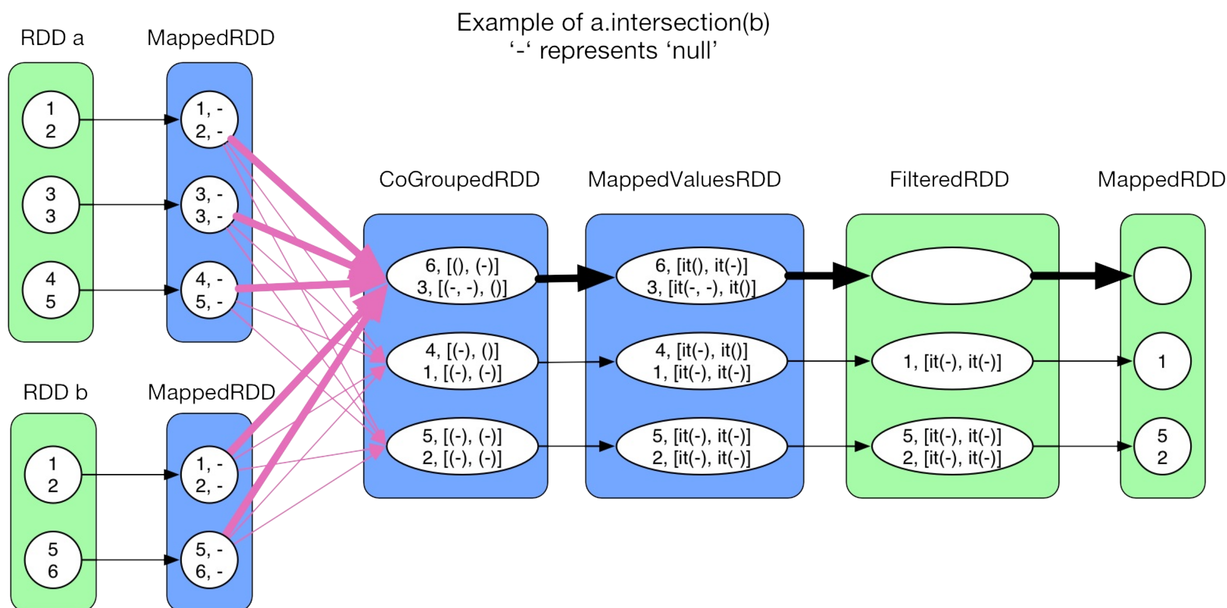


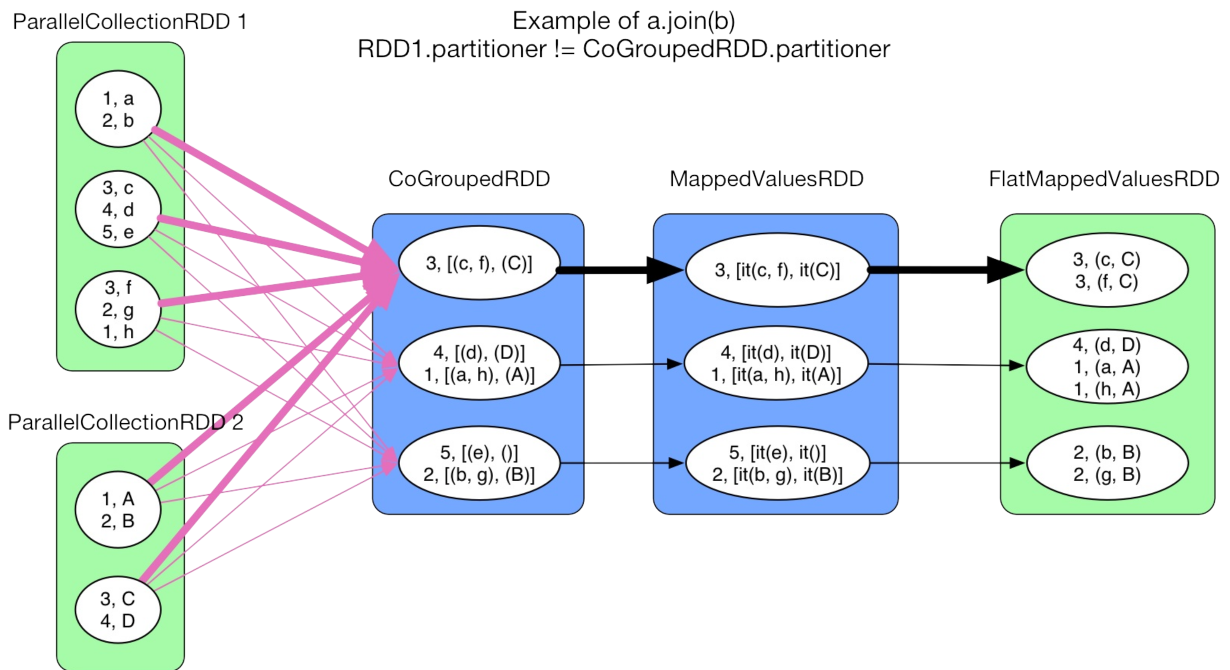
CoGroupedRDD 可能有 0 个、1 个或者多个 ShuffleDependency。但并不是要为每一个 ShuffleDependency 建立一个 HashMap，而是所有的 Dependency 共用一个 HashMap。与 reduceByKey() 不同的是，HashMap 在 CoGroupedRDD 的 compute() 中建立，而不是在 mapPartitionsWithContext() 中建立。

粗线表示的 task 首先 new 出一个 Array[ArrayBuffer(), ArrayBuffer()]，ArrayBuffer() 的个数与参与 cogroup 的 RDD 个数相同。func 的逻辑是这样的：每当从 RDD a 中 shuffle 过来一个 \ record 就将其添加到 hashmap.get(Key) 对应的 Array 中的第一个 ArrayBuffer() 中，每当从 RDD b 中 shuffle 过来一个 record，就将其添加到对应的 Array 中的第二个 ArrayBuffer()。

CoGroupedRDD => MappedValuesRDD 对应 mapValues() 操作，就是将 [ArrayBuffer(), ArrayBuffer()] 变成 [Iterable[V], Iterable[W]]。

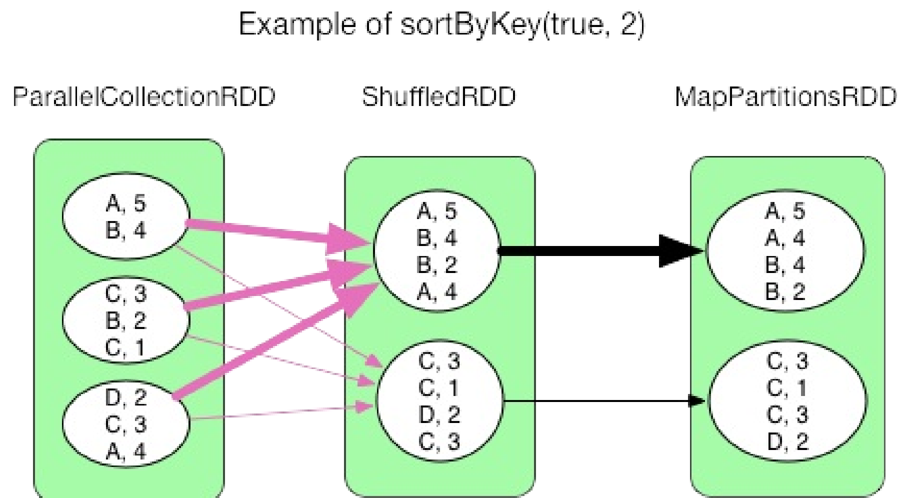
5. intersection(otherRDD) 和 join(otherRDD, numPartitions)





这两个操作中均使用了 cogroup，所以 shuffle 的处理方式与 cogroup 一样。

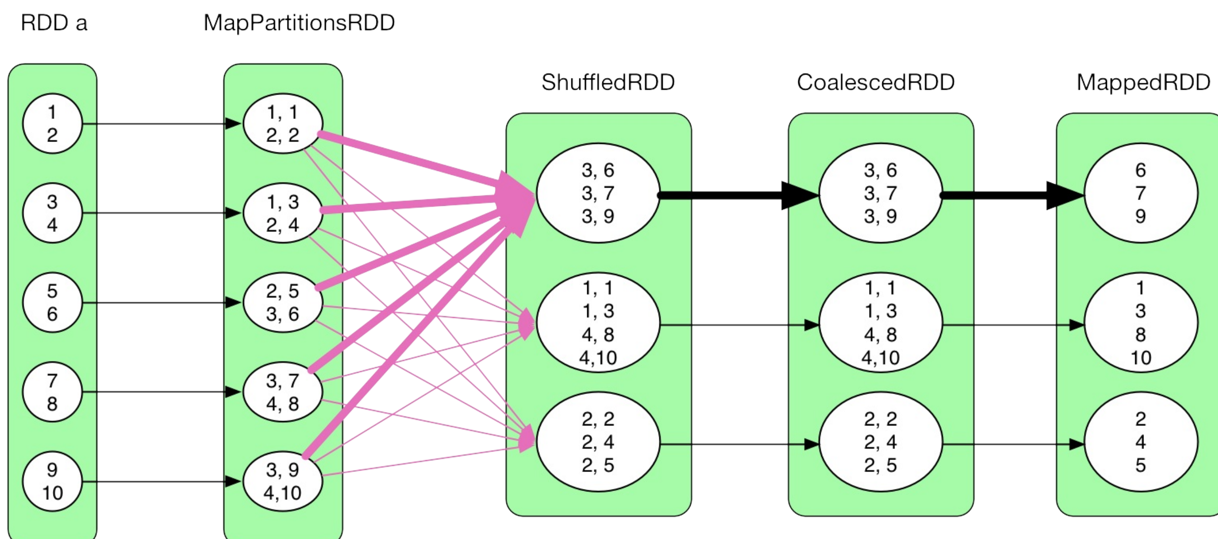
6. sortByKey(ascending, numPartitions)



sortByKey() 中 ShuffledRDD => MapPartitionsRDD 的处理逻辑与 reduceByKey() 不太一样，没有使用 HashMap 和 func 来处理 fetch 过来的 records。

sortByKey() 中 ShuffledRDD => MapPartitionsRDD 的处理逻辑是：将 shuffle 过来的一个个 record 存放到一个 Array 里，然后按照 Key 来对 Array 中的 records 进行 sort。

7. coalesce(numPartitions, shuffle = true)



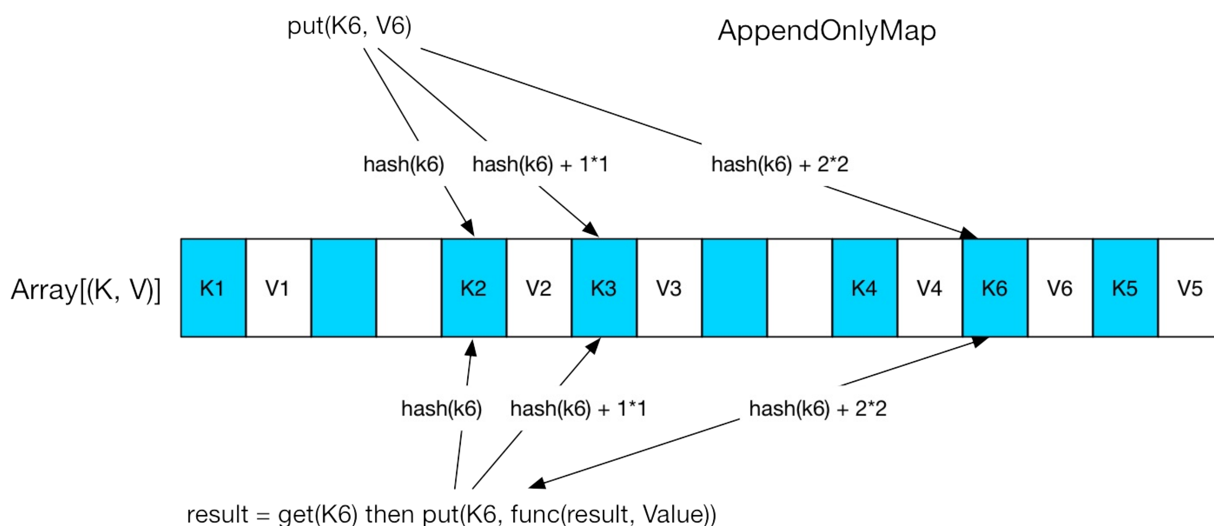
coalesce() 虽然有 ShuffleDependency，但不需要对 shuffle 过来的 records 进行 aggregate，所以没有建立 HashMap。每 shuffle 一个 record，就直接流向 CoalescedRDD，进而流向 MappedRDD 中。

Shuffle read 中的 HashMap

HashMap 是 Spark shuffle read 过程中频繁使用的、用于 aggregate 的数据结构。Spark 设计了两种：一种是全内存的 AppendOnlyMap，另一种是内存＋磁盘的 ExternalAppendOnlyMap。下面我们来分析一下两者特性及内存使用情况。

1. AppendOnlyMap

AppendOnlyMap 的官方介绍是 A simple open hash table optimized for the append-only use case, where keys are never removed, but the value for each key may be changed。意思是类似 HashMap，但没有 `remove(key)` 方法。其实现原理很简单，开一个大 Object 数组，蓝色部分存储 Key，白色部分存储 Value。如下图：



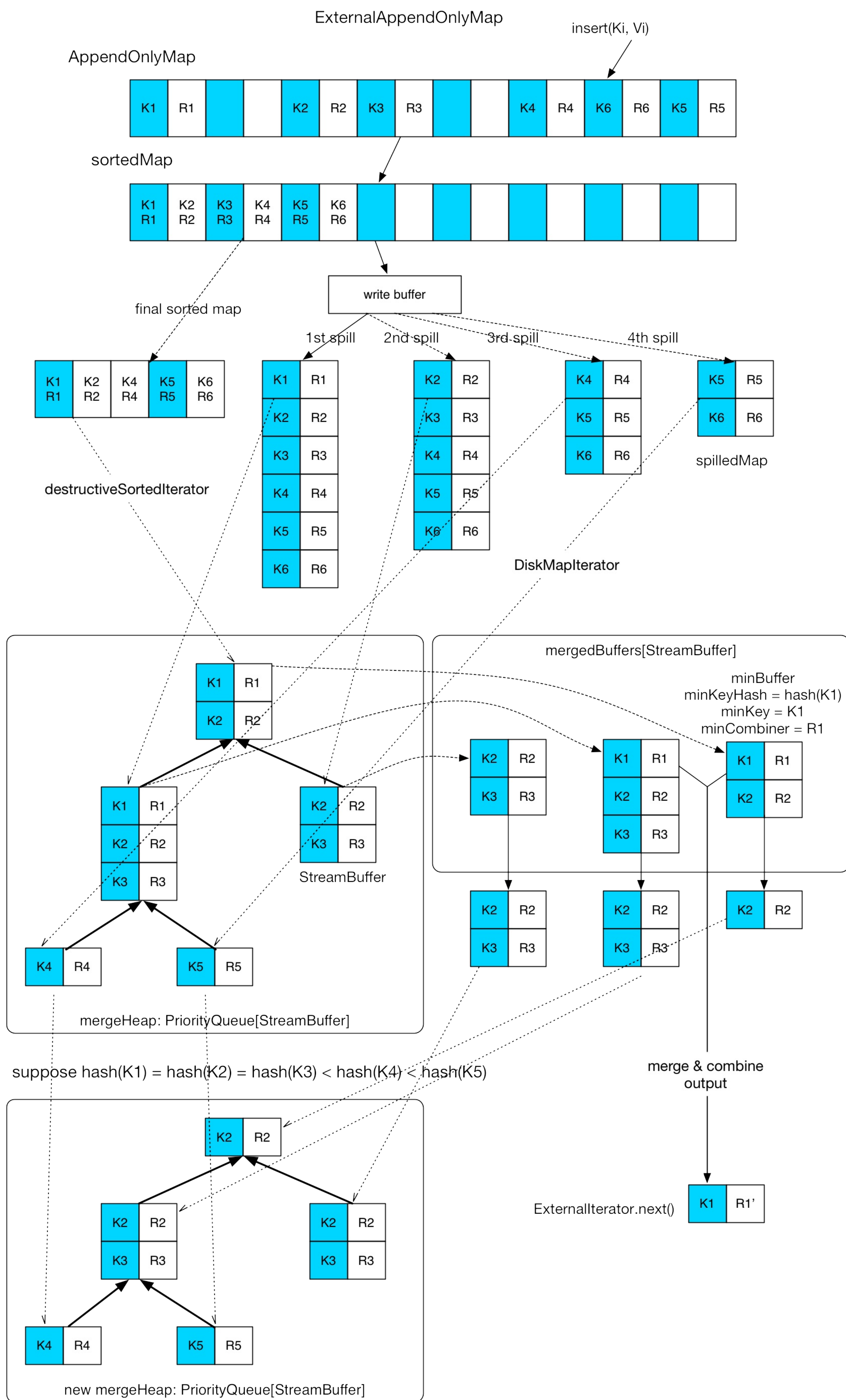
当要 put(K, V) 时，先 hash(K) 找存放位置，如果存放位置已经被占用，就使用 **Quadratic probing** 探测方法来找下一个空闲位置。对于图中的 K6 来说，第三次查找找到 K4 后面的空闲位置，放进去即可。get(K6) 的时候类似，找三次找到 K6，取出紧挨着的 V6，与先来的 value 做 func，结果重新放到 V6 的位置。

迭代 AppendOnlyMap 中的元素的时候，从前到后扫描输出。

如果 Array 的利用率达到 70%，那么就扩张一倍，并对所有 key 进行 rehash 后，重新排列每个 key 的位置。

AppendOnlyMap 还有一个 `destructiveSortedIterator(): Iterator[(K, V)]` 方法，可以返回 Array 中排序后的 (K, V) pairs。实现方法很简单：先将所有 (K, V) pairs compact 到 Array 的前端，并使得每个 (K, V) 占一个位置（原来占两个），之后直接调用 Array.sort() 排序，不过这样做会破坏数组（key 的位置变化了）。

2. ExternalAppendOnlyMap



ExternalAppendOnlyMap
insert(Ki, Vi)
AppendOnlyMap
sortedMap
write buffer
final sorted map
1st spill
2nd spill
3rd spill
4th spill
spilledMap
destructiveSortedIterator
DiskMapIterator
mergedBuffers[StreamBuffer]
minBuffer
minKeyHash = hash(K1)
minKey = K1
minCombiner = R1
StreamBuffer
mergeHeap: PriorityQueue[StreamBuffer]
suppose hash(K1) = hash(K2) = hash(K3) < hash(K4) < hash(K5)
merge & combine
output
ExternalIterator.next()
new mergeHeap: PriorityQueue[StreamBuffer]

相比 AppendOnlyMap，ExternalAppendOnlyMap 的实现略复杂，但逻辑其实很简单，类似 Hadoop MapReduce 中的 shuffle-merge-combine-sort 过程：

ExternalAppendOnlyMap 持有一个 AppendOnlyMap，shuffle 来的一个个 (K, V) record 先 insert 到 AppendOnlyMap 中，insert 过程与原始的 AppendOnlyMap 一模一样。如果 **AppendOnlyMap** 快被装满时检查一下内存剩余空间是否可以够扩展，够就直接在内存中扩展，不够就 **sort** 一下 **AppendOnlyMap**，将其内部所有 **records** 都 **spill** 到磁盘上。图中 spill 了 4 次，每次 spill 完在磁盘上生成一个 spilledMap 文件，然后重新 new 出来一个 AppendOnlyMap。最后一个 (K, V) record insert 到 AppendOnlyMap 后，表示所有 shuffle 来的 records 都被放到了 ExternalAppendOnlyMap 中，但不表示 records 已经被处理完，因为每次 insert 的时候，新来的 record 只与 AppendOnlyMap 中的 records 进行 aggregate，并不是与所有的 records 进行 aggregate（一些 records 已经被 spill 到磁盘上了）。因此当需要 aggregate 的最终结果时，需要对 AppendOnlyMap 和所有的 spilledMaps 进行全局 merge-aggregate。

全局 **merge-aggregate** 的流程也很简单：先将 AppendOnlyMap 中的 records 进行 sort，形成 sortedMap。然后利用 DestructiveSortedIterator 和 DiskMapIterator 分别从 sortedMap 和各个 spilledMap 读出一部分数据（StreamBuffer）放到 mergeHeap 里面。StreamBuffer 里面包含的 records 需要具有相同的 hash(key)，所以图中第一个 spilledMap 只读出前三个 records 进入 StreamBuffer。mergeHeap 顾名思义就是使用堆排序不断提取出 hash(firstRecord.Key) 相同的 StreamBuffer，并将其一个个放入 mergeBuffers 中，放入的时候与已经存在于 mergeBuffers 中的 StreamBuffer 进行 merge-combine，第一个被放入 mergeBuffers 的 StreamBuffer 被称为 minBuffer，那么 minKey 就是 minBuffer 中第一个 record 的 key。当 merge-combine 的时候，与 minKey 相同的 records 被 aggregate 一起，然后输出。整个 merge-combine 在 mergeBuffers 中结束后，StreamBuffer 剩余的 records 随着 StreamBuffer 重新进入 mergeHeap。一旦某个 StreamBuffer 在 merge-combine 后变为空（里面的 records 都被输出了），那么会使用 DestructiveSortedIterator 或 DiskMapIterator 重新装填 hash(key) 相同的 records，然后再重新进入 mergeHeap。

整个 insert-merge-aggregate 的过程有三点需要进一步探讨一下：

- 内存剩余空间检测

  与 Hadoop MapReduce 规定 reducer 中 70% 的空间可用于 shuffle-sort 类似，Spark 也规定 executor 中 `spark.shuffle.memoryFraction * spark.shuffle.safetyFraction` 的空间（默认是 `0.3 * 0.8`）可用于 ExternalOnlyAppendMap。**Spark** 略保守是不是？更保守的是这 **24**% 的空间不是完全用于一个 **ExternalOnlyAppendMap** 的，而是由在 **executor** 上同时运行的所有 **reducer** 共享的。为此，exectuor 专门持有一个 `ShuffleMemroyMap: HashMap[threadId, occupiedMemory]` 来监控每个 reducer 中 ExternalOnlyAppendMap 占用的内存量。每当 AppendOnlyMap 要扩展时，都会计算 **ShuffleMemroyMap** 持有的所有 **reducer** 中的 **AppendOnlyMap** 已占用的内存 ＋ 扩展后的内存 是会否会大于内存限制，大于就会将 AppendOnlyMap spill 到磁盘。有一点需要注意的是前 1000 个 records 进入 AppendOnlyMap 的时候不会启动是否要 **spill** 的检查，需要扩展时就直接在内存中扩展。

- AppendOnlyMap 大小估计

  为了获知 AppendOnlyMap 占用的内存空间，可以在每次扩展时都将 AppendOnlyMap reference 的所有 objects 大小都算一遍，然后加和，但这样做非常耗时。所以 Spark 设计了粗略的估算算法，算法时间复杂度是 O(1)，核心思想是利用 AppendOnlyMap 中每次 insert-aggregate record 后 result 的大小变化及一共 insert 的 records 的个数来估算大小，具体见 `SizeTrackingAppendOnlyMap` 和 `SizeEstimator`。

- Spill 过程

  与 shuffle write 一样，在 spill records 到磁盘上的时候，会建立一个 buffer 缓冲区，大小仍为 `spark.shuffle.file.buffer.kb`，默认是 32KB。另外，由于 serializer 也会分配缓冲区用于序列化和反序列化，所以如果一次 serialize 的 records 过多的话缓冲区会变得很大。Spark 限制每次 serialize 的 records 个数为 `spark.shuffle.spill.batchSize`，默认是 10000。

Discussion

通过本章的介绍可以发现，相比 MapReduce 固定的 shuffle-combine-merge-reduce 策略，Spark 更加灵活，会根据不同的

transformation() 的语义去设计不同的 shuffle-aggregate 策略，再加上不同的内存数据结构来混搭出合理的执行流程。

这章主要讨论了 Spark 是怎么在不排序 records 的情况下完成 shuffle write 和 shuffle read，以及怎么将 shuffle 过程融入 RDD computing chain 中的。附带讨论了内存与磁盘的平衡以及与 Hadoop MapReduce shuffle 的异同。下一章将从部署图以及进程通信角度来描述 job 执行的整个流程，也会涉及 shuffle write 和 shuffle read 中的数据位置获取问题。

另外，Jerry Shao 写的 详细探究Spark的shuffle实现 很赞，里面还介绍了 shuffle 过程在 Spark 中的进化史。目前 sort-based 的 shuffle 也在实现当中，stay tuned。

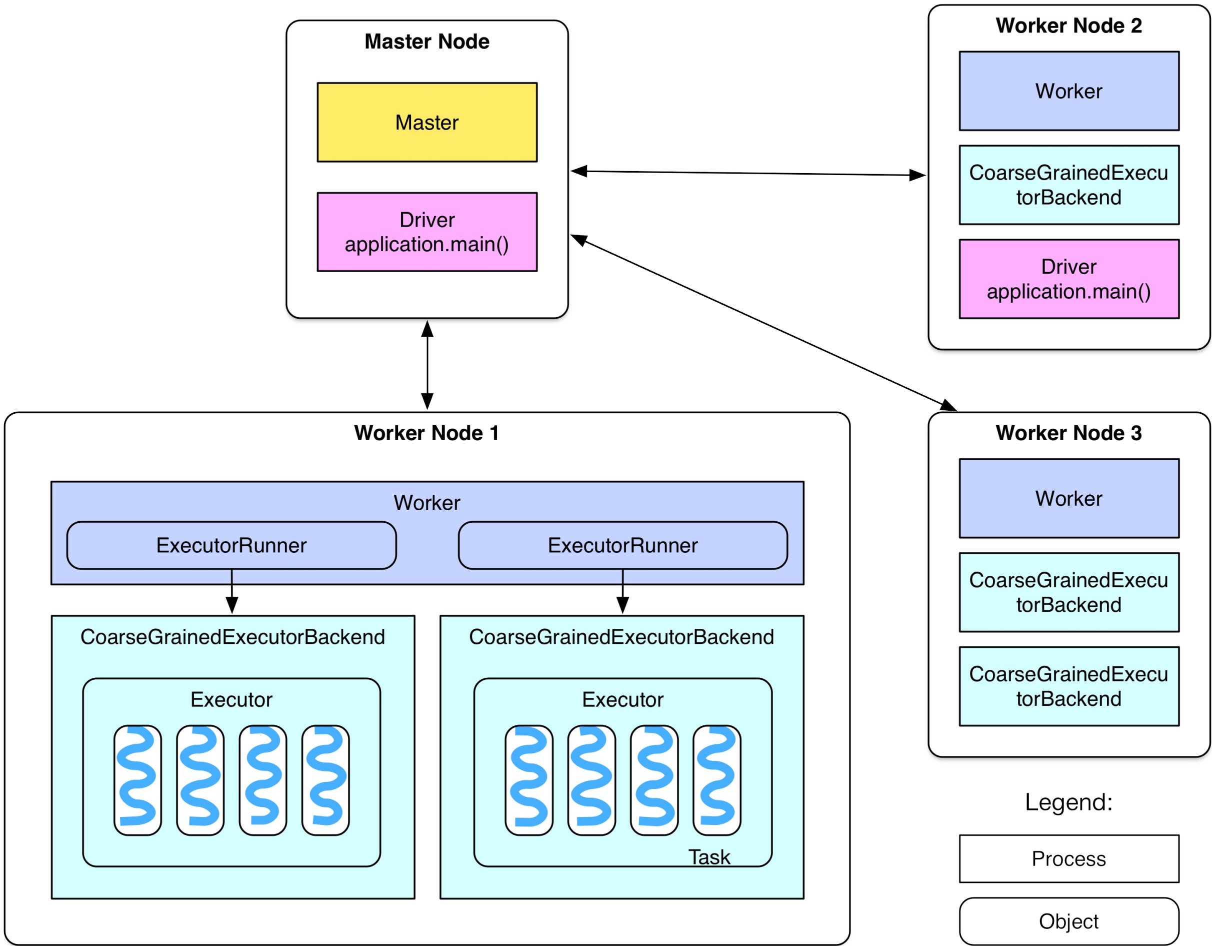
架构

前三章从 job 的角度介绍了用户写的 program 如何一步步地被分解和执行。这一章主要从架构的角度来讨论 master, worker，driver 和 executor 之间怎么协调来完成整个 job 的运行。

> 实在不想在文档中贴过多的代码，这章贴这么多，只是为了方面自己回头 debug 的时候可以迅速定位，不想看代码的话，直接看图和描述即可。

部署图

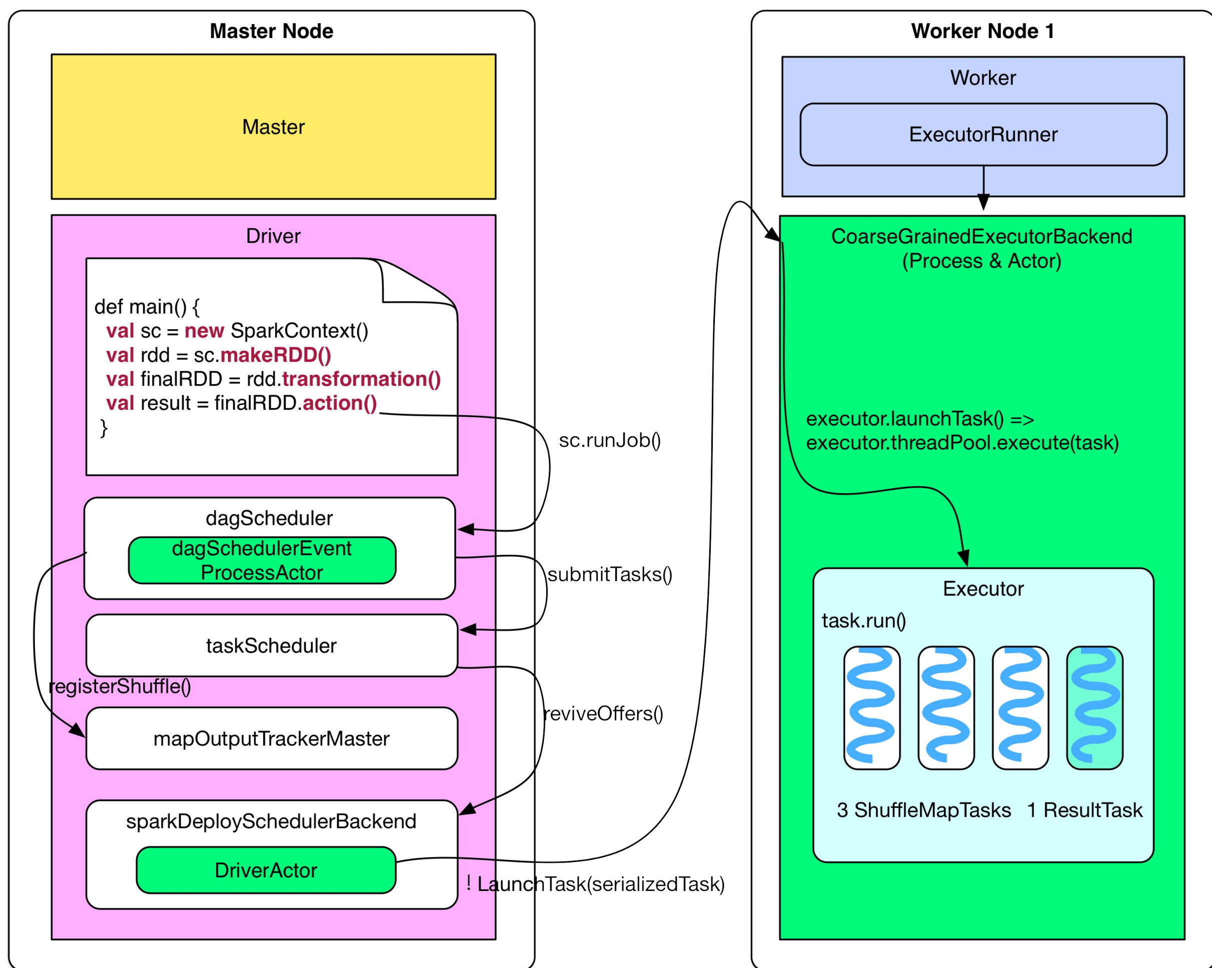
重新贴一下 Overview 中给出的部署图：



接下来分阶段讨论并细化这个图。

Job 提交

下图展示了driver program（假设在 master node 上运行）如何生成 job，并提交到 worker node 上执行。



Driver 端的逻辑如果用代码表示：

```
finalRDD.action()
=> sc.runJob()

// generate job, stages and tasks
=> dagScheduler.runJob()
=> dagScheduler.submitJob()
=>   dagSchedulerEventProcessActor ! JobSubmitted
=> dagSchedulerEventProcessActor.JobSubmitted()
=> dagScheduler.handleJobSubmitted()
=> finalStage = newStage()
=>   mapOutputTracker.registerShuffle(shuffleId, rdd.partitions.size)
=> dagScheduler.submitStage()
=>   missingStages = dagScheduler.getMissingParentStages()
=> dagScheduler.subMissingTasks(readyStage)

// add tasks to the taskScheduler
=> taskScheduler.submitTasks(new TaskSet(tasks))
=> fifoSchedulableBuilder.addTaskSetManager(taskSet)

// send tasks
=> sparkDeploySchedulerBackend.reviveOffers()
=> driverActor ! ReviveOffers
=> sparkDeploySchedulerBackend.makeOffers()
=> sparkDeploySchedulerBackend.launchTasks()
=> foreach task
      CoarseGrainedExecutorBackend(executorId) ! LaunchTask(serializedTask)
```

代码的文字描述：

当用户的 program 调用 `val sc = new SparkContext(sparkConf)` 时，这个语句会帮助 program 启动诸多有关 driver 通信、job 执行的对象、线程、actor等，该语句确立了 **program** 的 **driver** 地位。

生成 Job 逻辑执行图

Driver program 中的 transformation() 建立 computing chain（一系列的 RDD），每个 RDD 的 compute() 定义数据来了怎么计算得到该 RDD 中 partition 的结果，getDependencies() 定义 RDD 之间 partition 的数据依赖。

生成 Job 物理执行图

每个 action() 触发生成一个 job，在 dagScheduler.runJob() 的时候进行 stage 划分，在 submitStage() 的时候生成该 stage 包含的具体的 ShuffleMapTasks 或者 ResultTasks，然后将 tasks 打包成 TaskSet 交给 taskScheduler，如果 taskSet 可以运行就将 tasks 交给 sparkDeploySchedulerBackend 去分配执行。

分配 Task

sparkDeploySchedulerBackend 接收到 taskSet 后，会通过自带的 DriverActor 将 serialized tasks 发送到调度器指定的 worker node 上的 CoarseGrainedExecutorBackend Actor上。

Job 接收

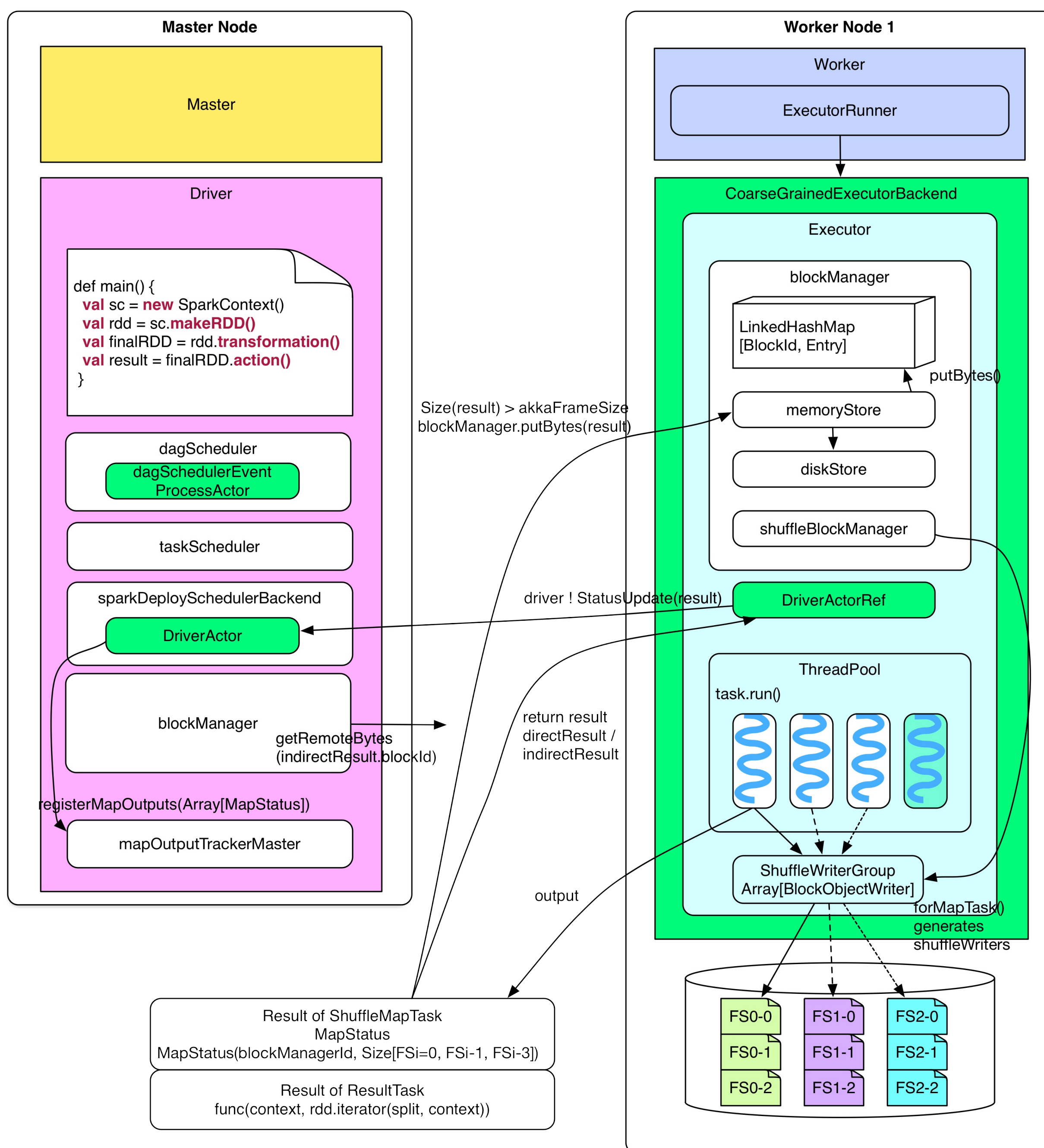
Worker 端接收到 tasks 后，执行如下操作

```
coarseGrainedExecutorBackend ! LaunchTask(serializedTask)
=> executor.launchTask()
=> executor.threadPool.execute(new TaskRunner(taskId, serializedTask))
```

executor 将 **task** 包装成 **taskRunner**，并从线程池中抽取出一个空闲线程运行 **task**。一个 **CoarseGrainedExecutorBackend** 进程有且仅有一个 **executor** 对象。

Task 运行

下图展示了 task 被分配到 worker node 上后的执行流程及 driver 如何处理 task 的 result。



Executor 收到 serialized 的 task 后，先 deserialize 出正常的 task，然后运行 task 得到其执行结果 directResult，这个结果要送回到 driver 那里。但是通过 Actor 发送的数据包不易过大，如果 **result** 比较大（比如 **groupByKey** 的 **result**）先把 **result** 存放到本地的“内存＋磁盘”上，由 **blockManager** 来管理，只把存储位置信息（**indirectResult**）发送给 **driver**，driver 需要实际的 result 的时候，会通过 HTTP 去 fetch。如果 result 不大（小于 `spark.akka.frameSize = 10MB`），那么直接发送给 driver。

上面的描述还有一些细节：如果 task 运行结束生成的 directResult > akka.frameSize，directResult 会被存放到由 blockManager 管理的本地“内存＋磁盘”上。**BlockManager** 中的 **memoryStore** 开辟了一个 **LinkedHashMap** 来存储要存放到本地内存的数据。LinkedHashMap 存储的数据总大小不超过 `Runtime.getRuntime.maxMemory * spark.storage.memoryFraction(default 0.6)`。如果 LinkedHashMap 剩余空间不足以存放新来的数据，就将数据交给 diskStore 存放到磁盘上，但前提是该数据的 storageLevel 中包含“磁盘”。

```
In TaskRunner.run()
// deserialize task, run it and then send the result to
=> coarseGrainedExecutorBackend.statusUpdate()
```

```
=> task = ser.deserialize(serializedTask)
=> value = task.run(taskId)
=> directResult = new DirectTaskResult(ser.serialize(value))
=> if( directResult.size() > akkaFrameSize() )
       indirectResult = blockManager.putBytes(taskId, directResult, MEMORY+DISK+SER)
   else
       return directResult
=> coarseGrainedExecutorBackend.statusUpdate(result)
=> driver ! StatusUpdate(executorId, taskId, result)
```

ShuffleMapTask 和 ResultTask 生成的 result 不一样。**ShuffleMapTask** 生成的是 **MapStatus**，MapStatus 包含两项内容：一是该 task 所在的 BlockManager 的 BlockManagerId（实际是 executorId + host, port, nettyPort），二是 task 输出的每个 FileSegment 大小。**ResultTask** 生成的 **result** 的是 **func** 在 **partition** 上的执行结果。比如 count() 的 func 就是统计 partition 中 records 的个数。由于 ShuffleMapTask 需要将 FileSegment 写入磁盘，因此需要输出流 writers，这些 writers 是由 blockManger 里面的 shuffleBlockManager 产生和控制的。

```
In task.run(taskId)
// if the task is ShuffleMapTask
=> shuffleMapTask.runTask(context)
=> shuffleWriterGroup = shuffleBlockManager.forMapTask(shuffleId, partitionId, numOutputSplits)
=> shuffleWriterGroup.writers(bucketId).write(rdd.iterator(split, context))
=> return MapStatus(blockManager.blockManagerId, Array[compressedSize(fileSegment)])

//If the task is ResultTask
=> return func(context, rdd.iterator(split, context))
```

Driver 收到 task 的执行结果 result 后会进行一系列的操作：首先告诉 taskScheduler 这个 task 已经执行完，然后去分析 result。由于 result 可能是 indirectResult，需要先调用 blockManager.getRemoteBytes() 去 fech 实际的 result，这个过程下节会详解。得到实际的 result 后，需要分情况分析，如果是 **ResultTask** 的 **result**，那么可以使用 **ResultHandler** 对 **result** 进行 **driver** 端的计算（比如 **count()** 会对所有 **ResultTask** 的 **result** 作 **sum**），如果 result 是 ShuffleMapTask 的 MapStatus，那么需要将 MapStatus（ShuffleMapTask 输出的 FileSegment 的位置和大小信息）存放到 **mapOutputTrackerMaster** 中的 **mapStatuses** 数据结构中以便以后 **reducer shuffle** 的时候查询。如果 driver 收到的 task 是该 stage 中的最后一个 task，那么可以 submit 下一个 stage，如果该 stage 已经是最后一个 stage，那么告诉 dagScheduler job 已经完成。

```
After driver receives StatusUpdate(result)
=> taskScheduler.statusUpdate(taskId, state, result.value)
=> taskResultGetter.enqueueSuccessfulTask(taskSet, tid, result)
=> if result is IndirectResult
      serializedTaskResult = blockManager.getRemoteBytes(IndirectResult.blockId)
=> scheduler.handleSuccessfulTask(taskSetManager, tid, result)
=> taskSetManager.handleSuccessfulTask(tid, taskResult)
=> dagScheduler.taskEnded(result.value, result.accumUpdates)
=> dagSchedulerEventProcessActor ! CompletionEvent(result, accumUpdates)
=> dagScheduler.handleTaskCompletion(completion)
=> Accumulators.add(event.accumUpdates)

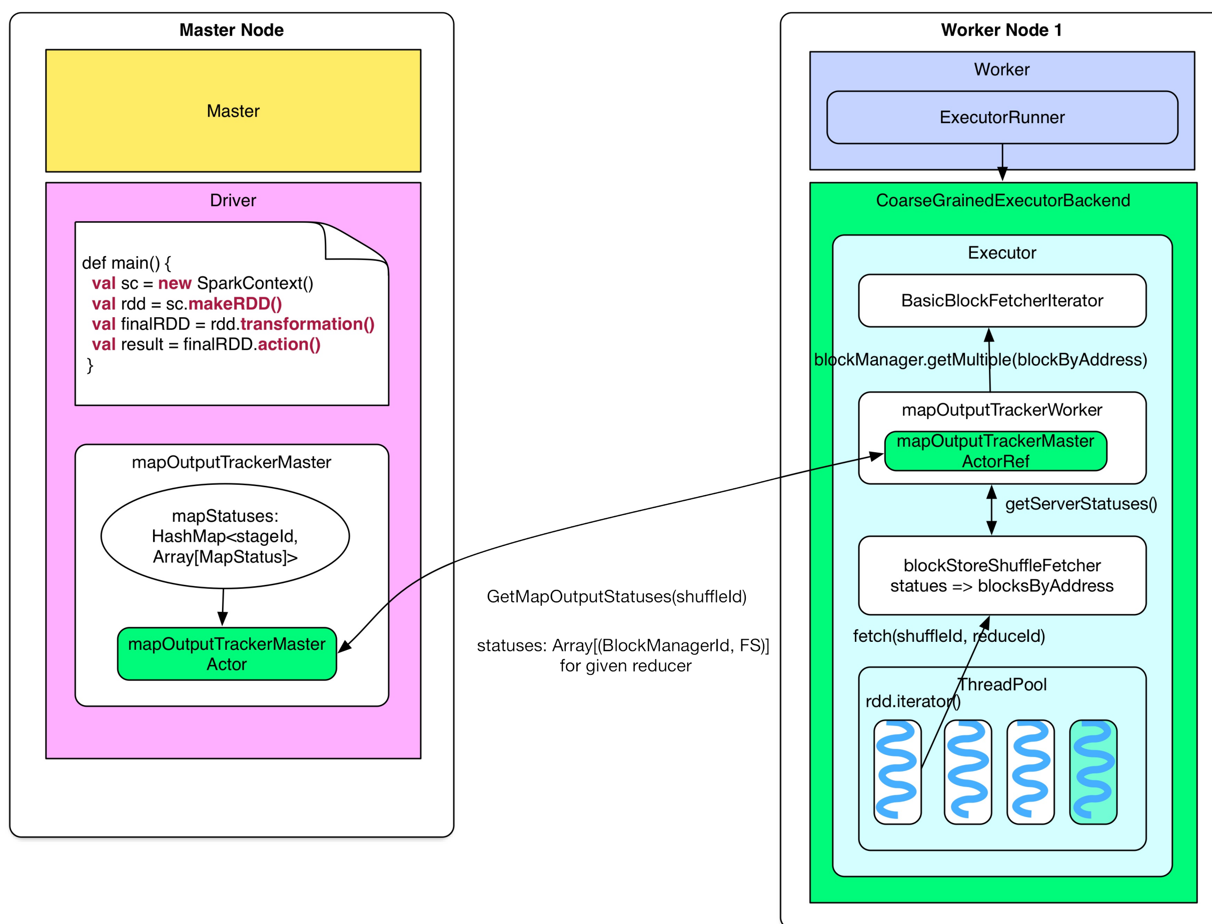
// If the finished task is ResultTask
=> if (job.numFinished == job.numPartitions)
      listenerBus.post(SparkListenerJobEnd(job.jobId, JobSucceeded))
=> job.listener.taskSucceeded(outputId, result)
=>    jobWaiter.taskSucceeded(index, result)
=>    resultHandler(index, result)

// if the finished task is ShuffleMapTask
=> stage.addOutputLoc(smt.partitionId, status)
=> if (all tasks in current stage have finished)
      mapOutputTrackerMaster.registerMapOutputs(shuffleId, Array[MapStatus])
      mapStatuses.put(shuffleId, Array[MapStatus]() ++ statuses)
=> submitStage(stage)
```

Shuffle read

上一节描述了 task 运行过程及 result 的处理过程，这一节描述 reducer（需要 shuffle 的 task ）是如何获取到输入数据的。关于 reducer 如何处理输入数据已经在上一章的 shuffle read 中解释了。

问题：**reducer 怎么知道要去哪里 fetch 数据？**

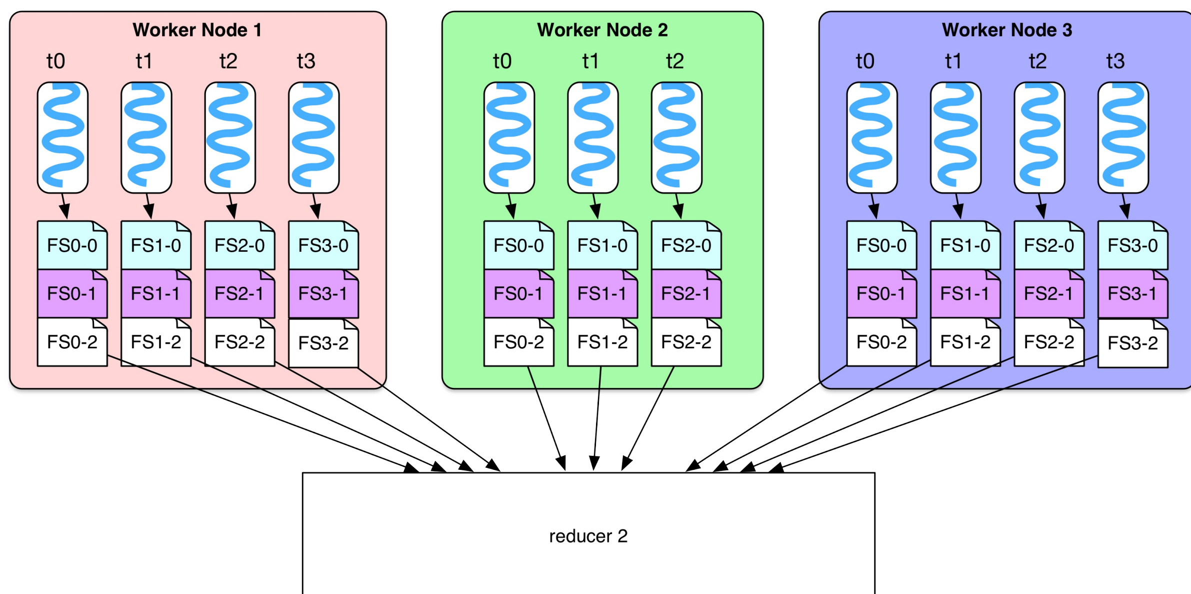


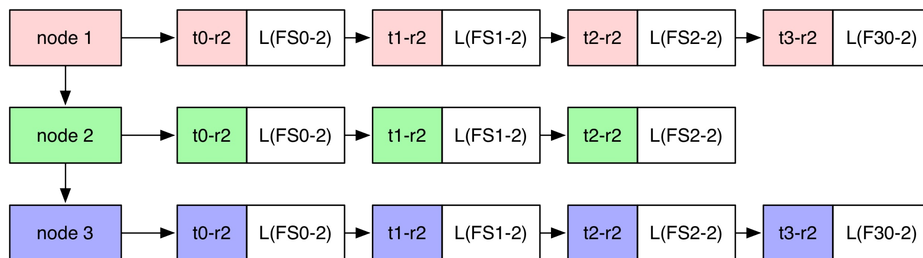
reducer 首先要知道 parent stage 中 ShuffleMapTask 输出的 FileSegments 在哪个节点。这个信息在 **ShuffleMapTask** 完成时已经送到了 **driver** 的 **mapOutputTrackerMaster**，并存放到了 **mapStatuses: HashMap** 里面，给定 stageId，可以获取该 stage 中 ShuffleMapTasks 生成的 FileSegments 信息 Array[MapStatus]，通过 Array(taskId) 就可以得到某个 task 输出的 FileSegments 位置（blockManagerId）及每个 FileSegment 大小。

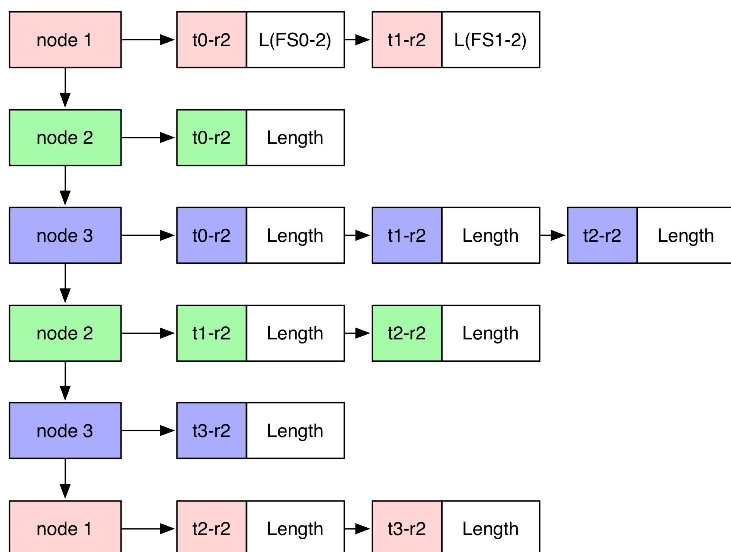
当 reducer 需要 fetch 输入数据的时候，会首先调用 blockStoreShuffleFetcher 去获取输入数据（FileSegments）的位置。blockStoreShuffleFetcher 通过调用本地的 MapOutputTrackerWorker 去完成这个任务，MapOutputTrackerWorker 使用 mapOutputTrackerMasterActorRef 来与 mapOutputTrackerMasterActor 通信获取 MapStatus 信息。blockStoreShuffleFetcher 对获取到的 MapStatus 信息进行加工，提取出该 reducer 应该去哪些节点上获取哪些 FileSegment 的信息，这个信息存放在 blocksByAddress 里面。之后，blockStoreShuffleFetcher 将获取 FileSegment 数据的任务交给 basicBlockFetcherIterator。

```
rdd.iterator()
=> rdd(e.g., ShuffledRDD/CoGroupedRDD).compute()
=> SparkEnv.get.shuffleFetcher.fetch(shuffledId, split.index, context, ser)
=> blockStoreShuffleFetcher.fetch(shuffleId, reduceId, context, serializer)
=> statuses = MapOutputTrackerWorker.getServerStatuses(shuffleId, reduceId)

=> blocksByAddress: Seq[(BlockManagerId, Seq[(BlockId, Long)])] = compute(statuses)
=> basicBlockFetcherIterator = blockManager.getMultiple(blocksByAddress, serializer)
=> itr = basicBlockFetcherIterator.flatMap(unpackBlock)
```







basicBlockFetcherIterator 收到获取数据的任务后，会生成一个个 fetchRequest，每个 **fetchRequest** 包含去某个节点获取若干个 **FileSegments** 的任务。图中展示了 reducer-2 需要从三个 worker node 上获取所需的白色 FileSegment (FS)。总的数据获取任务由 blocksByAddress 表示，要从第一个 node 获取 4 个，从第二个 node 获取 3 个，从第三个 node 获取 4 个。

为了加快任务获取过程，显然要将总任务划分为子任务（fetchRequest），然后为每个任务分配一个线程去 fetch。Spark 为

每个 reducer 启动 5 个并行 fetch 的线程（Hadoop 也是默认启动 5 个）。由于 fetch 来的数据会先被放到内存作缓冲，因此一次 fetch 的数据不能太多，Spark 设定不能超过 `spark.reducer.maxMbInFlight＝48MB`。注意这 **48MB** 的空间是由这 **5** 个 **fetch** 线程共享的，因此在划分子任务时，尽量使得 fetchRequest 不超过`48MB / 5 = 9.6MB`。如图在 node 1 中，Size(FS0-2) + Size(FS1-2) < 9.6MB 但是 Size(FS0-2) + Size(FS1-2) + Size(FS2-2) > 9.6MB，因此要在 t1-r2 和 t2-r2 处断开，所以图中有两个 fetchRequest 都是要去 node 1 fetch。那么会不会有 **fetchRequest** 超过 **9.6MB**？当然会有，如果某个 FileSegment 特别大，仍然需要一次性将这个 FileSegment fetch 过来。另外，如果 reducer 需要的某些 FileSegment 就在本节点上，那么直接进行 local read。最后，将 fetch 来的 FileSegment 进行 deserialize，将里面的 records 以 iterator 的形式提供给 rdd.compute()，整个 shuffle read 结束。

```
In basicBlockFetcherIterator:

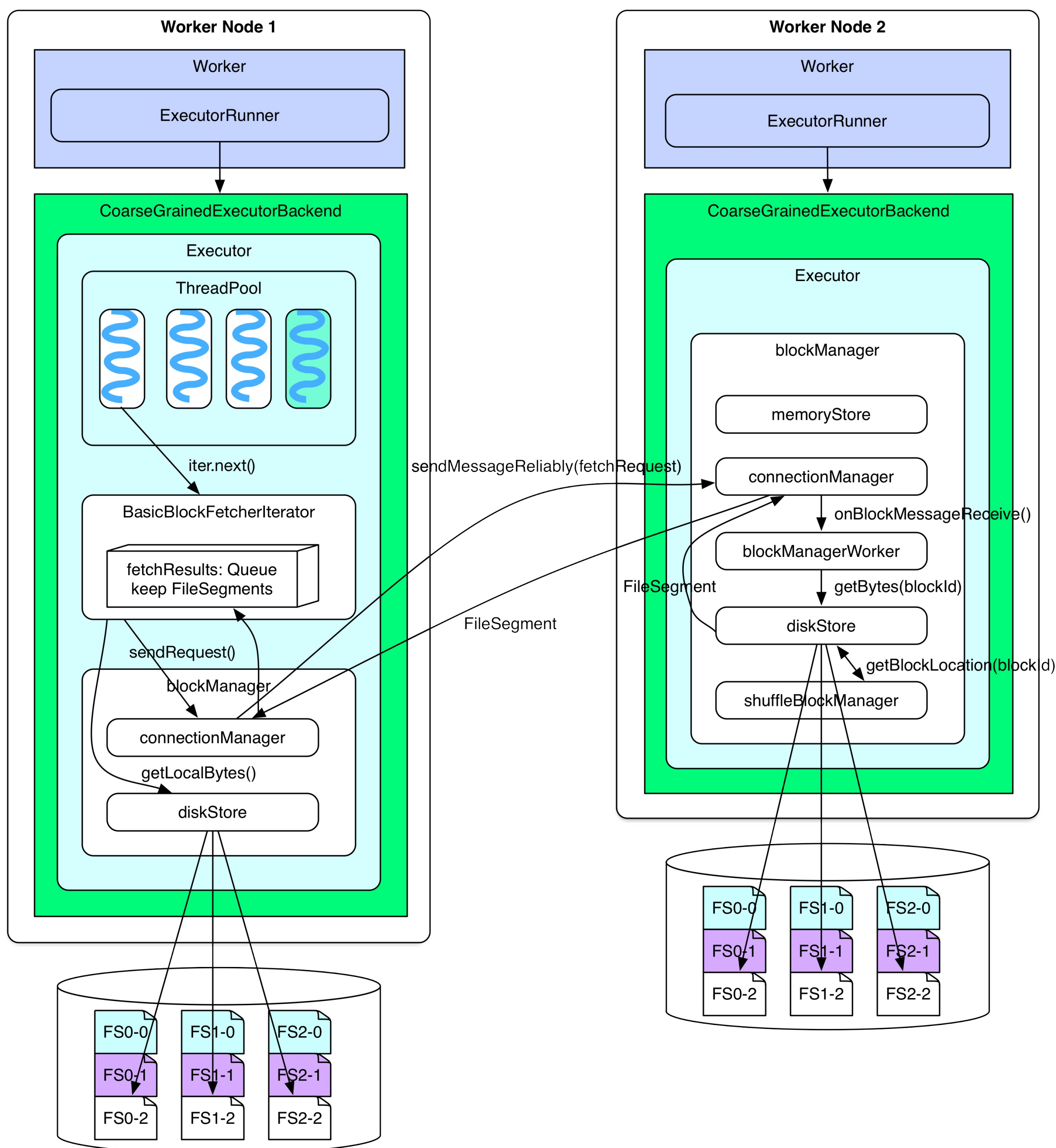
// generate the fetch requests
=> basicBlockFetcherIterator.initialize()
=> remoteRequests = splitLocalRemoteBlocks()
=> fetchRequests ++= Utils.randomize(remoteRequests)

// fetch remote blocks
=> sendRequest(fetchRequests.dequeue()) until Size(fetchRequests) > maxBytesInFlight
=> blockManager.connectionManager.sendMessageReliably(cmId,
       blockMessageArray.toBufferMessage)
=> fetchResults.put(new FetchResult(blockId, sizeMap(blockId)))
=> dataDeserialize(blockId, blockMessage.getData, serializer)

// fetch local blocks
=> getLocalBlocks()
=> fetchResults.put(new FetchResult(id, 0, () => iter))
```

下面再讨论一些细节问题：

reducer 如何将 **fetchRequest** 信息发送到目标节点？目标节点如何处理 **fetchRequest** 信息，如何读取 **FileSegment** 并回送给 **reducer**？



rdd.iterator() 碰到 ShuffleDependency 时会调用 BasicBlockFetcherIterator 去获取 FileSegments。BasicBlockFetcherIterator 使用 blockManager 中的 connectionManager 将 fetchRequest 发送给其他节点的 connectionManager。connectionManager 之间使用 NIO 模式通信。其他节点，比如 worker node 2 上的 connectionManager 收到消息后，会交给 blockManagerWorker 处理，blockManagerWorker 使用 blockManager 中的 diskStore 去本地磁盘上读取 fetchRequest 要求的 FileSegments，然后仍然通过 connectionManager 将 FileSegments 发送回去。如果使用了 FileConsolidation，diskStore 还需要 shuffleBlockManager 来提供 blockId 所在的具体位置。如果 FileSegment 不超过 `spark.storage.memoryMapThreshold=8KB` ，那么 diskStore 在读取 FileSegment 的时候会直接将 FileSegment 放到内存中，否则，会使用 RandomAccessFile 中 FileChannel 的内存映射方法来读取 FileSegment（这样可以将大的 FileSegment 加载到内存）。

当 BasicBlockFetcherIterator 收到其他节点返回的 serialized FileSegments 后会将其放到 fetchResults: Queue 里面，并进行 deserialization，所以 **fetchResults: Queue** 就相当于在 **Shuffle details** 那一章提到的 **softBuffer**。如果 BasicBlockFetcherIterator 所需的某些 FileSegments 就在本地，会通过 diskStore 直接从本地文件读取，并放到 fetchResults 里面。最后 reducer 一边从 FileSegment 中边读取 records 一边处理。

```
After the blockManager receives the fetch request

=> connectionManager.receiveMessage(bufferMessage)
=> handleMessage(connectionManagerId, message, connection)

// invoke blockManagerWorker to read the block (FileSegment)
=> blockManagerWorker.onBlockMessageReceive()
=> blockManagerWorker.processBlockMessage(blockMessage)
=> buffer = blockManager.getLocalBytes(blockId)
=> buffer = diskStore.getBytes(blockId)
=> fileSegment = diskManager.getBlockLocation(blockId)
=> shuffleManager.getBlockLocation()
=> if(fileSegment < minMemoryMapBytes)
     buffer = ByteBuffer.allocate(fileSegment)
   else
     channel.map(MapMode.READ_ONLY, segment.offset, segment.length)
```

每个 reducer 都持有一个 BasicBlockFetcherIterator，一个 BasicBlockFetcherIterator 理论上可以持有 48MB 的 fetchResults。每当 fetchResults 中有一个 FileSegment 被读取完，就会一下子去 fetch 很多个 FileSegment，直到 48MB 被填满。

```
BasicBlockFetcherIterator.next()
=> result = results.task()
=> while (!fetchRequests.isEmpty &&
        (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <= maxBytesInFlight)) {
        sendRequest(fetchRequests.dequeue())
      }
=> result.deserialize()
```

Discussion

这一章写了三天，也是我这个月来心情最不好的几天。Anyway，继续总结。

架构部分其实没有什么好说的，就是设计时尽量功能独立，模块独立，松耦合。BlockManager 设计的不错，就是管的东西太多（数据块、内存、磁盘、通信）。

这一章主要探讨了系统中各个模块是怎么协同来完成 job 的生成、提交、运行、结果收集、结果计算以及 shuffle 的。贴了很多代码，也画了很多图，虽然细节很多，但远没有达到源码的细致程度。如果有地方不明白的，请根据描述阅读一下源码吧。

如果想进一步了解 blockManager，可以参阅 Jerry Shao 写的 Spark源码分析之-Storage模块。

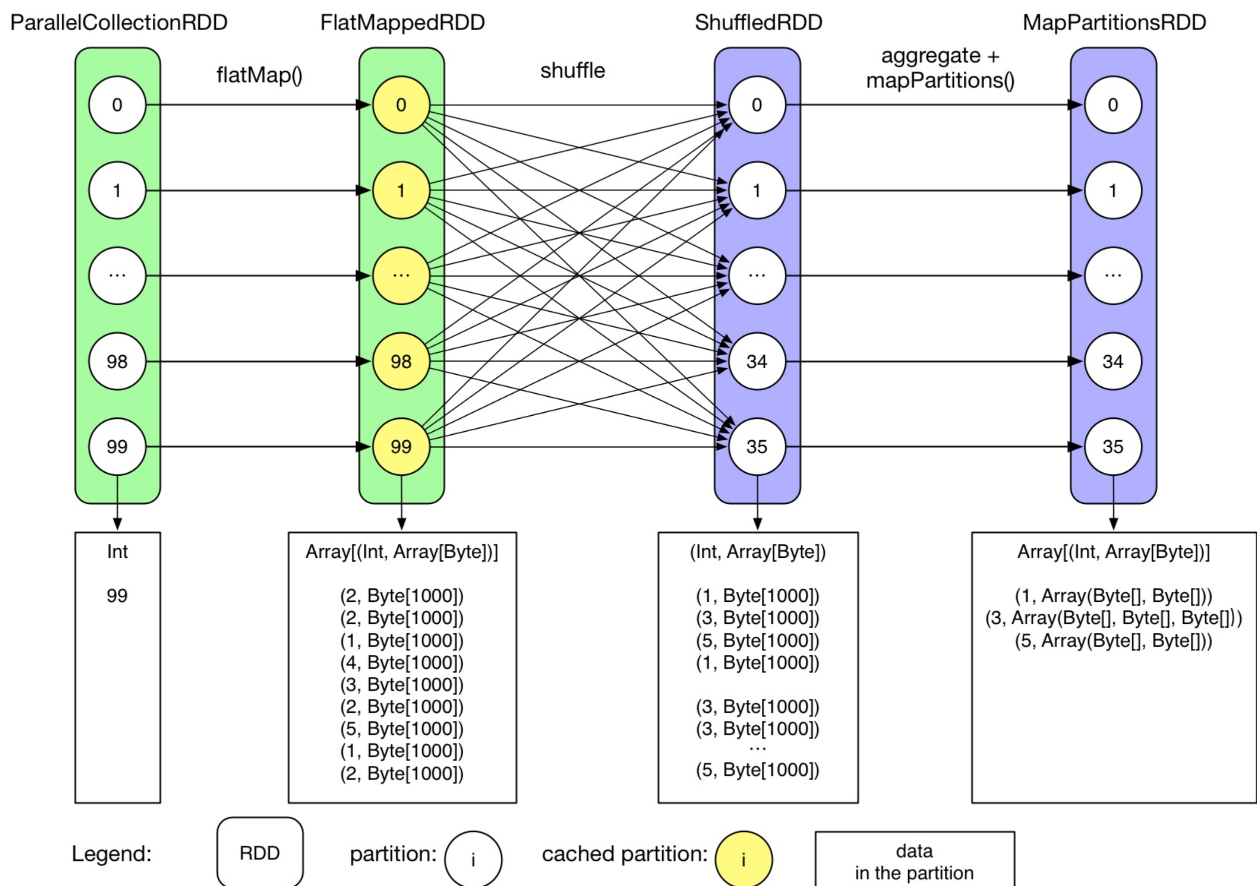
Cache 和 Checkpoint

作为区别于 Hadoop 的一个重要 feature，cache 机制保证了需要访问重复数据的应用（如迭代型算法和交互式应用）可以运行的更快。与 Hadoop MapReduce job 不同的是 Spark 的逻辑/物理执行图可能很庞大，task 中 computing chain 可能会很长，计算某些 RDD 也可能会很耗时。这时，如果 task 中途运行出错，那么 task 的整个 computing chain 需要重算，代价太高。因此，有必要将计算代价较大的 RDD checkpoint 一下，这样，当下游 RDD 计算出错时，可以直接从 checkpoint 过的 RDD 那里读取数据继续算。

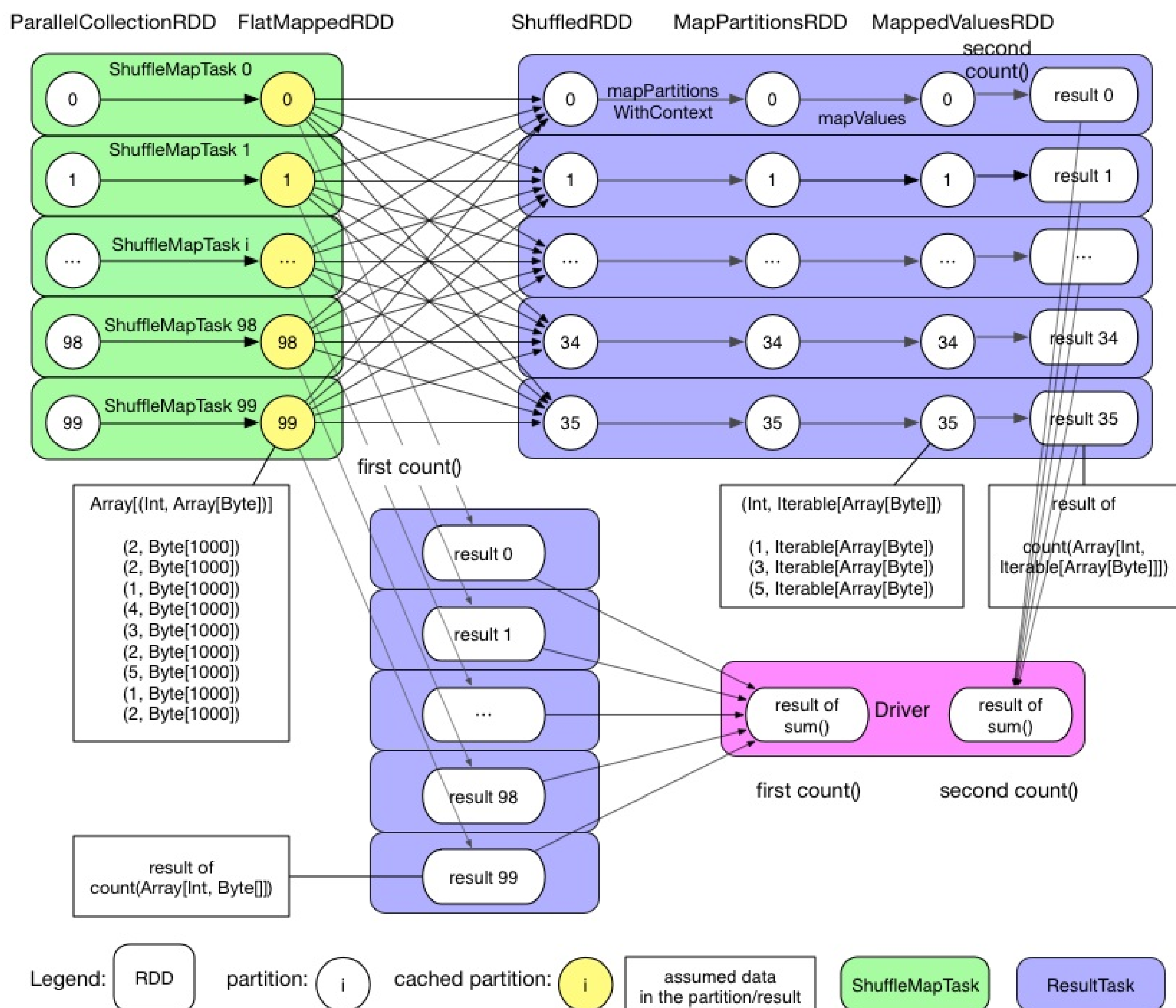
Cache 机制

回到 Overview 提到的 GroupByTest 的例子，里面对 FlatMappedRDD 进行了 cache，这样 Job 1 在执行时就直接从 FlatMappedRDD 开始算了。可见 cache 能够让重复数据在同一个 application 中的 jobs 间共享。

逻辑执行图：



物理执行图：



问题：哪些 **RDD** 需要 **cache**？

会被重复使用的（但不能太大）。

问题：用户怎么设定哪些 **RDD** 要 **cache**？

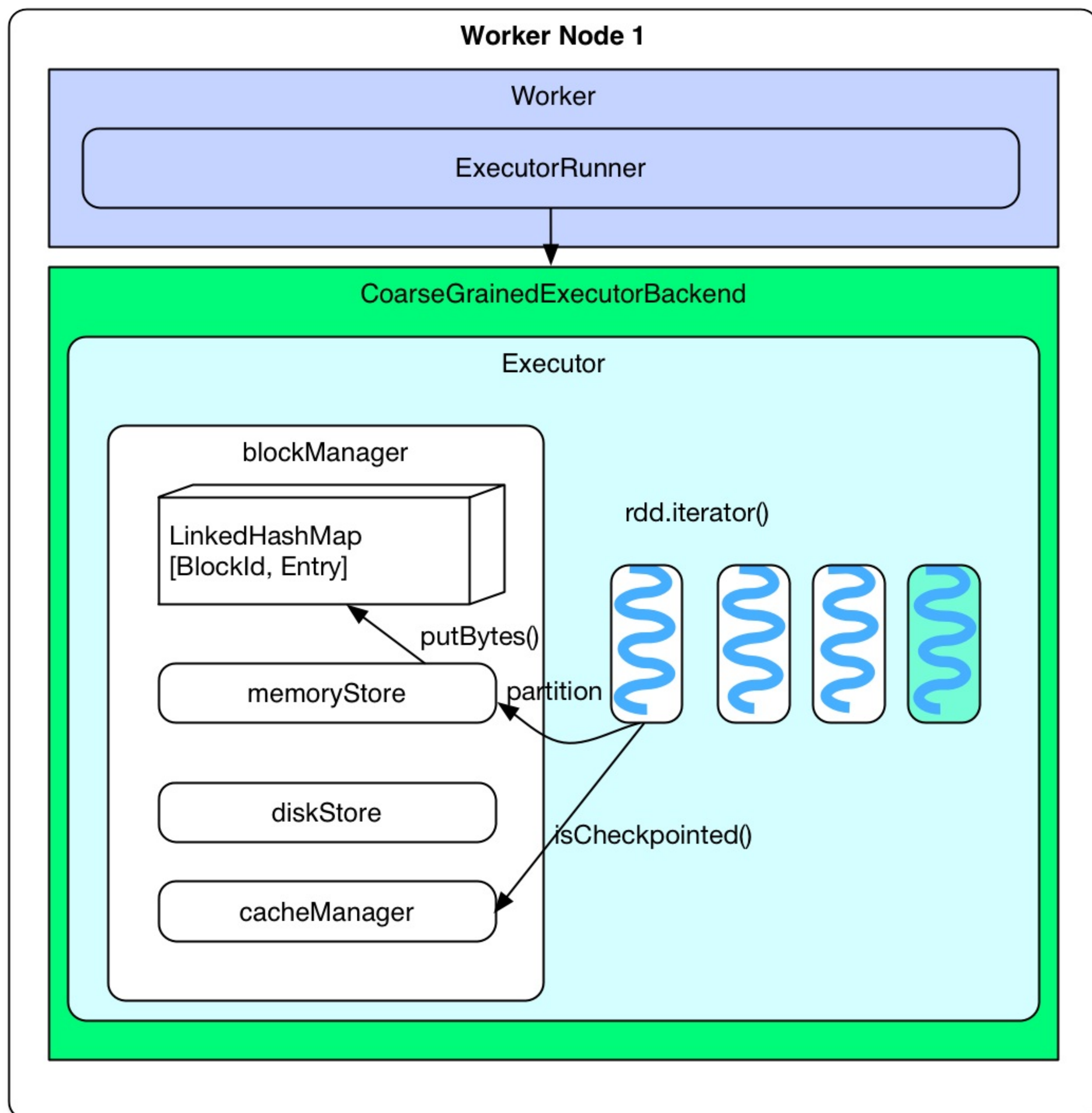
因为用户只与 driver program 打交道，因此只能用 rdd.cache() 去 cache 用户能看到的 RDD。所谓能看到指的是调用 transformation() 后生成的 RDD，而某些在 transformation() 中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，比如 reduceByKey() 中会生成的 ShuffledRDD、MapPartitionsRDD 是不能被用户直接 cache 的。

问题：**driver program** 设定 **rdd.cache()** 后，系统怎么对 **RDD** 进行 **cache**？

先不看实现，自己来想象一下如何完成 cache：当 task 计算得到 RDD 的某个 partition 的第一个 record 后，就去判断该 RDD 是否要被 cache，如果要被 cache 的话，将这个 record 及后续计算的到的 records 直接丢给本地 blockManager 的 memoryStore，如果 memoryStore 存不下就交给 diskStore 存放到磁盘。

实际实现与设想的基本类似，区别在于：将要计算 RDD partition 的时候（而不是已经计算得到第一个 record 的时候）就去判断 partition 要不要被 cache。如果要被 cache 的话，先将 partition 计算出来，然后 cache 到内存。cache 只使用 memory，写磁盘的话那就叫 checkpoint 了。

调用 rdd.cache() 后， rdd 就变成 persistRDD 了，其 StorageLevel 为 MEMORY_ONLY。persistRDD 会告知 driver 说自己是需要被 persist 的。



如果用代码表示：

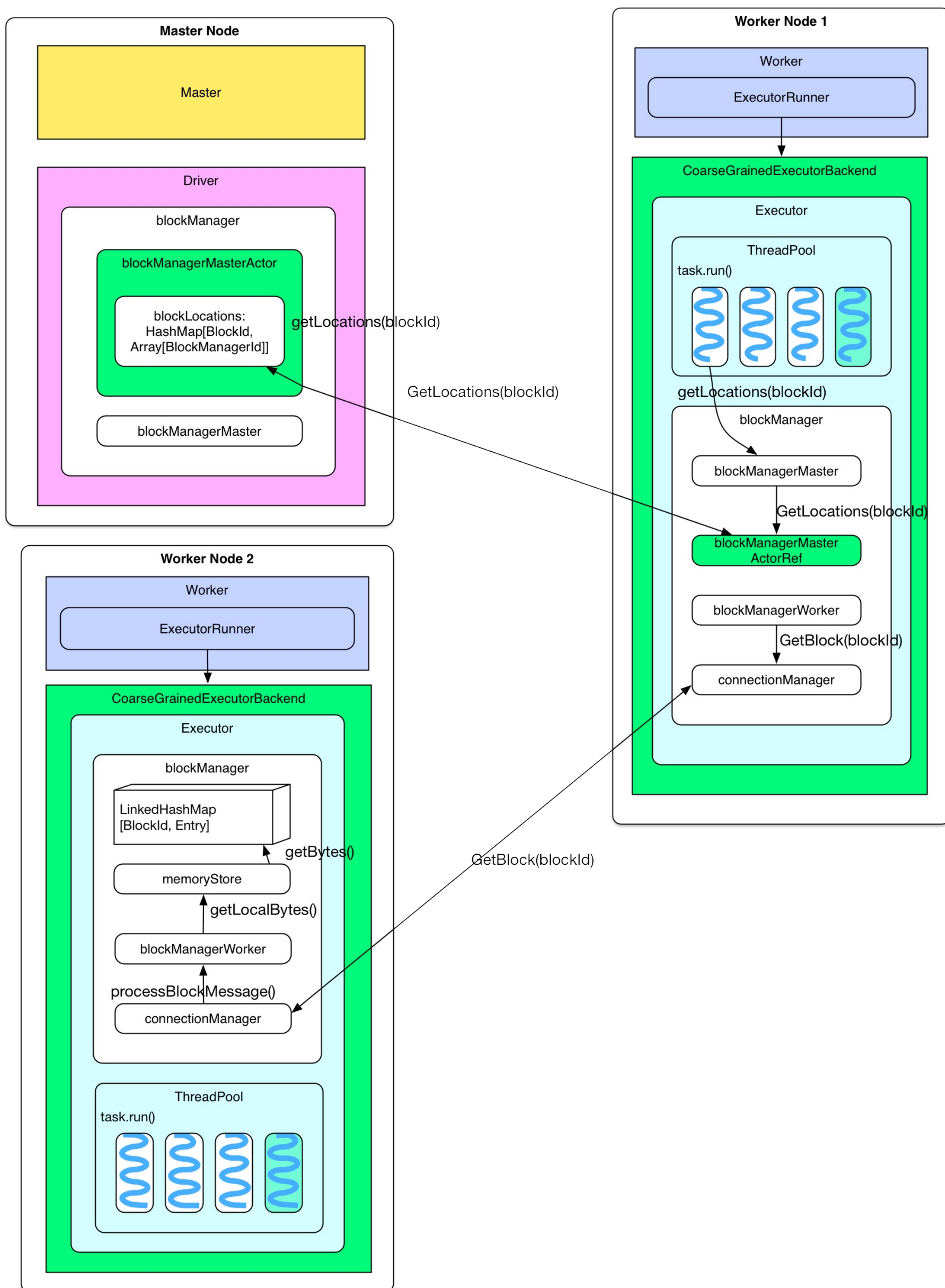
```
rdd.iterator()
=> SparkEnv.get.cacheManager.getOrCompute(thisRDD, split, context, storageLevel)
=> key = RDDBlockId(rdd.id, split.index)
=> blockManager.get(key)
=> computedValues = rdd.computeOrReadCheckpoint(split, context)
      if (isCheckpointed) firstParent[T].iterator(split, context)
      else compute(split, context)
=> elements = new ArrayBuffer[Any]
=> elements ++= computedValues
=> updatedBlocks = blockManager.put(key, elements, tellMaster = true)
```

当 rdd.iterator() 被调用的时候，也就是要计算该 rdd 中某个 partition 的时候，会先去 cacheManager 那里领取一个 blockId，表明是要存哪个 RDD 的哪个 partition，这个 blockId 类型是 RDDBlockId（memoryStore 里面可能还存放有 task 的 result 等数据，因此 blockId 的类型是用来区分不同的数据）。然后去 blockManager 里面查看该 partition 是不是已经被 checkpoint 了，如果是，表明以前运行过该 task，那就不用计算该 partition 了，直接从 checkpoint 中读取该 partition 的所有 records 放到叫做 elements 的 ArrayBuffer 里面。如果没有被 checkpoint 过，先将 partition 计算出来，然后将其所有 records 放到 elements 里面。最后将 elements 交给 blockManager 进行 cache。

blockManager 将 elements（也就是 partition） 存放到 memoryStore 管理的 LinkedHashMap[BlockId, Entry] 里面。如果 partition 大于 memoryStore 的存储极限（默认是 60% 的 heap），那么直接返回说存不下。如果剩余空间也许能放下，会先 drop 掉一些早先被 cached 的 RDD 的 partition，为新来的 partition 腾地方，如果腾出的地方够，就把新来的 partition 放到 LinkedHashMap 里面，腾不出就返回说存不下。注意 drop 的时候不会去 drop 与新来的 partition 同属于一个 RDD 的 partition。drop 的时候先 drop 最早被 cache 的 partition。（说好的 LRU 替换算法呢？）

问题：**cached RDD** 怎么被读取？

下次计算（一般是同一 application 的下一个 job 计算）时如果用到 cached RDD，task 会直接去 blockManager 的 memoryStore 中读取。具体地讲，当要计算某个 rdd 中的 partition 时候（通过调用 rdd.iterator()）会先去 blockManager 里面查找是否已经被 cache 了，如果 partition 被 cache 在本地，就直接使用 blockManager.getLocal() 去本地 memoryStore 里读取。如果该 partition 被其他节点上 blockManager cache 了，会通过 blockManager.getRemote() 去其他节点上读取，读取过程如下图。



获取 **cached partitions** 的存储位置：partition 被 cache 后所在节点上的 blockManager 会通知 driver 上的 blockMangerMasterActor 说某 rdd 的 partition 已经被我 cache 了，这个信息会存储在 blockMangerMasterActor 的 blockLocations: HashMap中。等到 task 执行需要 cached rdd 的时候，会调用 blockManagerMaster 的 getLocations(blockId) 去询问某 partition 的存储位置，这个询问信息会发到 driver 那里，driver 查询 blockLocations 获得位置信息并将信息送回。

读取其他节点上的 **cached partition**：task 得到 cached partition 的位置信息后，将 GetBlock(blockId) 的请求通过 connectionManager 发送到目标节点。目标节点收到请求后从本地 blockManager 那里的 memoryStore 读取 cached partition，最后发送回来。

Checkpoint

问题：哪些 **RDD** 需要 **checkpoint**？

运算时间很长或运算量太大才能得到的 RDD，computing chain 过长或依赖其他 RDD 很多的 RDD。 实际上，将 ShuffleMapTask 的输出结果存放到本地磁盘也算是 checkpoint，只不过这个 checkpoint 的主要目的是去 partition 输出数据。

问题：什么时候 **checkpoint**？

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。也就是说需要 **checkpoint** 的 **RDD** 会被计算两次。因此，在使用 **rdd.checkpoint()** 的时候，建议加上 **rdd.cache()**，这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。其实 Spark 提供了 rdd.persist(StorageLevel.DISK_ONLY) 这样的方法，相当于 cache 到磁盘上，这样可以做到 rdd 第一次被计算得到时就存储到磁盘上，但这个 persist 和 checkpoint 有很多不同，之后会讨论。

问题：**checkpoint** 怎么实现？

RDD 需要经过 [Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed] 这几个阶段才能被 checkpoint。

Initialized： 首先 driver program 需要使用 rdd.checkpoint() 去设定哪些 rdd 需要 checkpoint，设定后，该 rdd 就接受 RDDCheckpointData 管理。用户还要设定 checkpoint 的存储路径，一般在 HDFS 上。

marked for checkpointing：初始化后，RDDCheckpointData 会将 rdd 标记为 MarkedForCheckpoint。

checkpointing in progress：每个 job 运行结束后会调用 finalRdd.doCheckpoint()，finalRdd 会顺着 computing chain 回溯扫描，碰到要 checkpoint 的 RDD 就将其标记为 CheckpointingInProgress，然后将写磁盘（比如写 HDFS）需要的配置文件（如 core-site.xml 等）broadcast 到其他 worker 节点上的 blockManager。完成以后，启动一个 job 来完成 checkpoint（使用 `rdd.context.runJob(rdd, CheckpointRDD.writeToFile(path.toString, broadcastedConf))`）。

checkpointed：job 完成 checkpoint 后，将该 rdd 的 dependency 全部清掉，并设定该 rdd 状态为 checkpointed。然后，为该 **rdd** 强加一个依赖，设置该 **rdd** 的 **parent rdd** 为 **CheckpointRDD**，该 CheckpointRDD 负责以后读取在文件系统上的 checkpoint 文件，生成该 rdd 的 partition。

有意思的是我在 driver program 里 checkpoint 了两个 rdd，结果只有一个（下面的 result）被 checkpoint 成功，pairs2 没有被 checkpoint，也不知道是 bug 还是故意只 checkpoint 下游的 RDD：

```
val data1 = Array[(Int, Char)]((1, 'a'), (2, 'b'), (3, 'c'),
    (4, 'd'), (5, 'e'), (3, 'f'), (2, 'g'), (1, 'h'))
val pairs1 = sc.parallelize(data1, 3)

val data2 = Array[(Int, Char)]((1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'))
val pairs2 = sc.parallelize(data2, 2)

pairs2.checkpoint

val result = pairs1.join(pairs2)
result.checkpoint
```

问题：怎么读取 **checkpoint** 过的 **RDD**？

在 runJob() 的时候会先调用 finalRDD 的 partitions() 来确定最后会有多个 task。rdd.partitions() 会去检查（通过 RDDCheckpointData 去检查，因为它负责管理被 checkpoint 过的 rdd）该 rdd 是会否被 checkpoint 过了，如果该 rdd 已经被 checkpoint 过了，直接返回该 rdd 的 partitions 也就是 Array[Partition]。

当调用 rdd.iterator() 去计算该 rdd 的 partition 的时候，会调用 computeOrReadCheckpoint(split: Partition) 去查看该 rdd 是否被 checkpoint 过了，如果是，就调用该 rdd 的 parent rdd 的 iterator() 也就是 CheckpointRDD.iterator()，CheckpointRDD 负责读取文件系统上的文件，生成该 rdd 的 partition。这就解释了为什么那么 **trickly** 地为 **checkpointed rdd** 添加一个 **parent CheckpointRDD**。

问题：**cache** 与 **checkpoint** 的区别？

关于这个问题，Tathagata Das 有一段回答: There is a significant difference between cache and checkpoint. Cache materializes the RDD and keeps it in memory and/or disk（其实只有 memory）. But the lineage（也就是 computing chain） of RDD (that is, seq of operations that generated the RDD) will be remembered, so that if there are node failures and parts of the cached RDDs are lost, they can be regenerated. However, **checkpoint saves the RDD to an HDFS file and actually forgets the lineage completely.** This is allows long lineages to be truncated and the data to be saved reliably in HDFS (which is naturally fault tolerant by replication).

深入一点讨论，rdd.persist(StorageLevel.DISK_ONLY) 与 checkpoint 也有区别。前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉（话说怎么 **remove checkpoint** 过的 **RDD**？），是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 dirver program 使用。

Discussion

Hadoop MapReduce 在执行 job 的时候，不停地做持久化，每个 task 运行结束做一次，每个 job 运行结束做一次（写到 HDFS）。在 task 运行过程中也不停地在内存和磁盘间 swap 来 swap 去。 可是讽刺的是，Hadoop 中的 task 太傻，中途出错需要完全重新运行，比如 shuffle 了一半的数据存放到了磁盘，下次重新运行时仍然要重新 shuffle。Spark 好的一点在于尽量不去持久化，所以使用 pipeline，cache 等机制。用户如果感觉 job 可能会出错可以手动去 checkpoint 一些 critical 的 RDD，job 如果出错，下次运行时直接从 checkpoint 中读取数据。唯一不足的是，checkpoint 需要两次运行 job。

Example

貌似还没有发现官方给出的 checkpoint 的例子，这里我写了一个：

```
package internals

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object groupByKeyTest {

   def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("GroupByKey").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setCheckpointDir("/Users/xulijie/Documents/data/checkpoint")

    val data = Array[(Int, Char)]((1, 'a'), (2, 'b'),
                                  (3, 'c'), (4, 'd'),
                                  (5, 'e'), (3, 'f'),
                                  (2, 'g'), (1, 'h')
                                 )
    val pairs = sc.parallelize(data, 3)

    pairs.checkpoint
```

```
    pairs.count

    val result = pairs.groupByKey(2)

    result.foreachWith(i => i)((x, i) => println("[PartitionIndex " + i + "] " + x))

    println(result.toDebugString)
   }
}
```

Broadcast

顾名思义，broadcast 就是将数据从一个节点发送到其他各个节点上去。这样的场景很多，比如 driver 上有一张表，其他节点上运行的 task 需要 lookup 这张表，那么 driver 可以先把这张表 copy 到这些节点，这样 task 就可以在本地查表了。如何实现一个可靠高效的 broadcast 机制是一个有挑战性的问题。先看看 Spark 官网上的一段话：

Broadcast variables allow the programmer to keep a **read-only** variable cached on each **machine** rather than shipping a copy of it with **tasks**. They can be used, for example, to give every node a copy of a **large input dataset** in an efficient manner. Spark also attempts to distribute broadcast variables using **efficient** broadcast algorithms to reduce communication cost.

问题：为什么只能 broadcast 只读的变量？

这就涉及一致性的问题，如果变量可以被更新，那么一旦变量被某个节点更新，其他节点要不要一块更新？如果多个节点同时在更新，更新顺序是什么？怎么做同步？还会涉及 fault-tolerance 的问题。为了避免维护数据一致性问题，Spark 目前只支持 broadcast 只读变量。

问题：broadcast 到节点而不是 broadcast 到每个 task？

因为每个 task 是一个线程，而且同在一个进程运行 tasks 都属于同一个 application。因此每个节点（executor）上放一份就可以被所有 task 共享。

问题： 具体怎么用 broadcast？

driver program 例子：

```
val data = List(1, 2, 3, 4, 5, 6)
val bdata = sc.broadcast(data)

val rdd = sc.parallelize(1 to 6, 2)
val observedSizes = rdd.map(_ => bdata.value.size)
```

driver 使用 `sc.broadcast()` 声明要 broadcast 的 data，bdata 的类型是 Broadcast。

当 `rdd.transformation(func)` 需要用 bdata 时，直接在 func 中调用，比如上面的例子中的 map() 就使用了 bdata.value.size。

问题：怎么实现 broadcast？

broadcast 的实现机制很有意思：

1. 分发 task 的时候先分发 bdata 的元信息

Driver 先建一个本地文件夹用以存放需要 broadcast 的 data，并启动一个可以访问该文件夹的 HttpServer。当调用 `val bdata = sc.broadcast(data)` 时就把 data 写入文件夹，同时写入 driver 自己的 blockManger 中（StorageLevel 为内存＋磁盘），获得一个 blockId，类型为 BroadcastBlockId。当调用 `rdd.transformation(func)` 时，如果 func 用到了 bdata，那么 driver submitTask() 的时候会将 bdata 一同 func 进行序列化得到 serialized task，注意序列化的时候不会序列化 **bdata 中包含的 data**。上一章讲到 serialized task 从 driverActor 传递到 executor 时使用 Akka 的传消息机制，消息不能太大，而实际的 data 可能很大，所以这时候还不能 broadcast data。

> driver 为什么会同时将 data 放到磁盘和 blockManager 里面？放到磁盘是为了让 HttpServer 访问到，放到 blockManager 是为了让 driver program 自身使用 bdata 时方便（其实我觉得不放到 blockManger 里面也行）。

那么什么时候传送真正的 **data**？在 executor 反序列化 task 的时候，会同时反序列化 task 中的 bdata 对象，这时候会调用 bdata 的 readObject() 方法。该方法先去本地 blockManager 那里询问 bdata 的 data 在不在 blockManager 里面，如果不在就使用下面的两种 fetch 方式之一去将 data fetch 过来。得到 data 后，将其存放到 blockManager 里面，这样后面运行的 task 如果需要 bdata 就不需要再去 fetch data 了。如果在，就直接拿来用了。

下面探讨 broadcast data 时候的两种实现方式：

2. HttpBroadcast

顾名思义，HttpBroadcast 就是每个 executor 通过的 http 协议连接 driver 并从 driver 那里 fetch data。

Driver 先准备好要 broadcast 的 data，调用 `sc.broadcast(data)` 后会调用工厂方法建立一个 HttpBroadcast 对象。该对象做的第一件事就是将 data 存到 driver 的 blockManager 里面，StorageLevel 为内存＋磁盘，blockId 类型为 BroadcastBlockId。

同时 driver 也会将 broadcast 的 data 写到本地磁盘，例如写入后得到 `/var/folders/87/grpn1_fn4xq5wdqmxk31v0l00000gp/T/spark-6233b09c-3c72-4a4d-832b-6c0791d0eb9c/broadcast_0`， 这个文件夹作为 HttpServer 的文件目录。

> Driver 和 executor 启动的时候，都会生成 broadcastManager 对象，调用 HttpBroadcast.initialize()，driver 会在本地建立一个临时目录用来存放 broadcast 的 data，并启动可以访问该目录的 httpServer。

Fetch data：在 executor 反序列化 task 的时候，会同时反序列化 task 中的 bdata 对象，这时候会调用 bdata 的 readObject() 方法。该方法先去本地 blockManager 那里询问 bdata 的 data 在不在 blockManager 里面，如果不在就使用 **http** 协议连接 **driver** 上的 **httpServer**，将 **data fetch** 过来。得到 data 后，将其存放到 blockManager 里面，这样后面运行的 task 如果需要 bdata 就不需要再去 fetch data 了。如果在，就直接拿来用了。

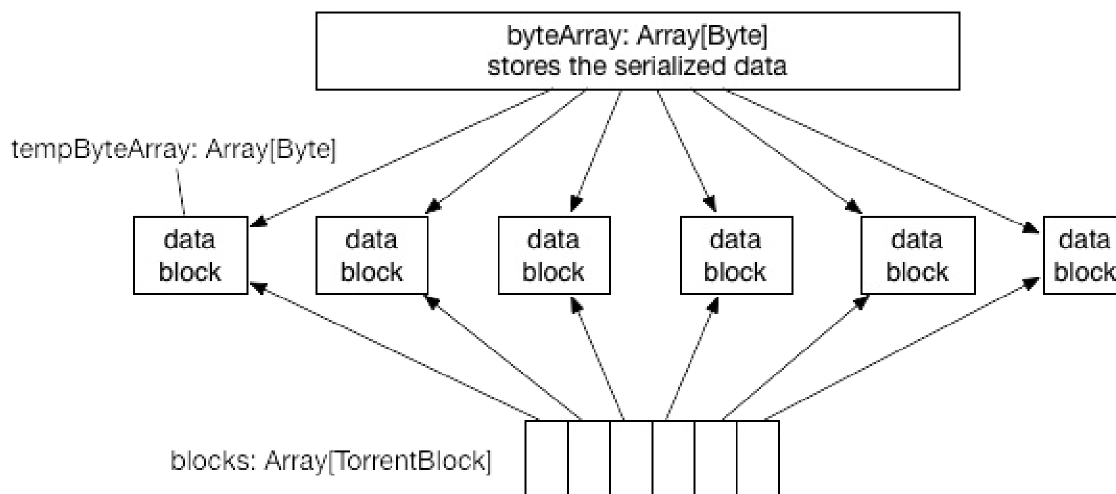
HttpBroadcast 最大的问题就是 **driver** 所在的节点可能会出现网络拥堵，因为 worker 上的 executor 都会去 driver 那里 fetch 数据。

3. TorrentBroadcast

为了解决 HttpBroadast 中 driver 单点网络瓶颈的问题，Spark 又设计了一种 broadcast 的方法称为 TorrentBroadcast，这个类似于大家常用的 **BitTorrent** 技术。基本思想就是将 data 分块成 data blocks，然后假设有 executor fetch 到了一些 data blocks，那么这个 executor 就可以被当作 data server 了，随着 fetch 的 executor 越来越多，有更多的 data server 加入，data 就很快能传播到全部的 executor 那里去了。

HttpBroadcast 是通过传统的 http 协议和 httpServer 去传 data，在 TorrentBroadcast 里面使用在上一章介绍的 blockManager.getRemote() => NIO ConnectionManager 传数据的方法来传递，读取数据的过程与读取 cached rdd 的方式类似，可以参阅 CacheAndCheckpoint 中的最后一张图。

下面讨论 TorrentBroadcast 的一些细节：



driver 端：

Driver 先把 data 序列化到 byteArray，然后切割成 BLOCK_SIZE（由 `spark.broadcast.blockSize = 4MB` 设置）大小的 data block，每个 data block 被 TorrentBlock 对象持有。切割完 byteArray 后，会将其回收，因此内存消耗虽然可以达到 2 * Size(data)，但这是暂时的。

完成分块切割后，就将分块信息（称为 meta 信息）存放到 driver 自己的 blockManager 里面，StorageLevel 为内存＋磁盘，同时会通知 driver 自己的 blockManagerMaster 说 meta 信息已经存放好。通知 **blockManagerMaster** 这一步很重要，因为 **blockManagerMaster** 可以被 **driver** 和所有 **executor** 访问到，信息被存放到 **blockManagerMaster** 就变成了全局信息。

之后将每个分块 data block 存放到 driver 的 blockManager 里面，StorageLevel 为内存＋磁盘。存放后仍然通知 blockManagerMaster 说 blocks 已经存放好。到这一步，driver 的任务已经完成。

Executor 端：

executor 收到 serialized task 后，先反序列化 task，这时候会反序列化 serialized task 中包含的 bdata 类型是 TorrentBroadcast，也就是去调用 TorrentBroadcast.readObject()。这个方法首先得到 bdata 对象，然后发现 **bdata** 里面没有包含实际的 **data**。怎么办？先询问所在的 executor 里的 blockManager 是会否包含 data（通过查询 data 的 broadcastId），包含就直接从本地 blockManager 读取 data。否则，就通过本地 blockManager 去连接 driver 的 blockManagerMaster 获取 data 分块的 meta 信息，获取信息后，就开始了 BT 过程。

BT 过程：task 先在本地开一个数组用于存放将要 fetch 过来的 data blocks `arrayOfBlocks = new Array[TorrentBlock](totalBlocks)`，TorrentBlock 是对 data block 的包装。然后打乱要 fetch 的 data blocks 的顺序，比如如果 data block 共有 5 个，那么打乱后的 fetch 顺序可能是 3-1-2-4-5。然后按照打乱后的顺序去 fetch 一个个 data block。fetch 的过程就是通过 “本地 blockManager －本地 connectionManager－driver/executor 的 connectionManager－driver/executor 的 blockManager－data” 得到 data，这个过程与 fetch cached rdd 类似。每 **fetch** 到一个 **block** 就将其存放到 **executor** 的 **blockManager** 里面，同时通知 **driver** 上的 **blockManagerMaster** 说该 **data block** 多了一个存储地址。这一步通知非常重要，意味着 blockManagerMaster 知道 data block 现在在 cluster 中有多份，下一个不同节点上的 task 再去 fetch 这个 data block 的时候，可以有两个选择了，而且会随机选择一个去 fetch。这个过程持续下去就是 BT 协议，随着下载的客户端越来越多，data block 服务器也越来越多，就变成 p2p下载了。关于 BT 协议，Wikipedia 上有一个动画)。

整个 fetch 过程结束后，task 会开一个大 Array[Byte]，大小为 data 的总大小，然后将 data block 都 copy 到这个 Array，然后对 Array 中 bytes 进行反序列化得到原始的 data，这个过程就是 driver 序列化 data 的反过程。

最后将 data 存放到 task 所在 executor 的 blockManager 里面，StorageLevel 为内存＋磁盘。显然，这时候 data 在 blockManager 里存了两份，不过等全部 executor 都 fetch 结束，存储 data blocks 那份可以删掉了。

问题：broadcast RDD 会怎样?

@Andrew-Xia 回答道：不会怎样，就是这个rdd在每个executor中实例化一份。

Discussion

公共数据的 broadcast 是很实用的功能，在 Hadoop 中使用 DistributedCache，比如常用的 `-libjars` 就是使用 DistributedCache 来将 task 依赖的 jars 分发到每个 task 的工作目录。不过分发前 DistributedCache 要先将文件上传到 HDFS。这种方式的主要问题是资源浪费，如果某个节点上要运行来自同一 job 的 4 个 mapper，那么公共数据会在该节点上存在 4 份（每个 task 的工作目录会有一份）。但是通过 HDFS 进行 broadcast 的好处在于单点瓶颈不明显，因为公共 data 首先被分成多个 block，然后不同的 block 存放在不同的节点。这样，只要所有的 task 不是同时去同一个节点 fetch 同一个 block，网络拥塞不会很严重。

对于 Spark 来讲，broadcast 时考虑的不仅是如何将公共 data 分发下去的问题，还要考虑如何让同一节点上的 task 共享 data。

对于第一个问题，Spark 设计了两种 broadcast 的方式，传统存在单点瓶颈问题的 HttpBroadcast，和类似 BT 方式的 TorrentBroadcast。HttpBroadcast 使用传统的 client-server 形式的 HttpServer 来传递真正的 data，而 TorrentBroadcast 使用 blockManager 自带的 NIO 通信方式来传递 data。TorrentBroadcast 存在的问题是慢启动和占内存，慢启动指的是刚开始 data 只在 driver 上有，要等 executors fetch 很多轮 data block 后，data server 才会变得可观，后面的 fetch 速度才会变快。executor 所占内存的在 fetch 完 data blocks 后进行反序列化时需要将近两倍 data size 的内存消耗。不管哪一种方式，driver 在分块时会有两倍 data size 的内存消耗。

对于第二个问题，每个 executor 都包含一个 blockManager 用来管理存放在 executor 里的数据，将公共数据存放在 blockManager 中（StorageLevel 为内存＋磁盘），可以保证在 executor 执行的 tasks 能够共享 data。

其实 Spark 之前还尝试了一种称为 TreeBroadcast 的机制，详情可以见技术报告 Performance and Scalability of Broadcast in Spark。

更深入点，broadcast 可以用多播协议来做，不过多播使用 UDP，不是可靠的，仍然需要应用层的设计一些可靠性保障机制。